C9

Thu Jan 03 12:25:21 2008 Function Index Page 2

Thu Jan 03 12:25:21 2008 Function Index Page 1

Thu Jan 03 12:25:21 2008 Function Index Page 2

Thu Jan 03 12:25:21 2008 File Index Page 1

Thu Jan 03 12:25:21 2008 File Index Page 2

```
/*****************************************************************************
                                                                          ******
**
** File Name:   RSLwisvr.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
** -----------
**      The intent of the contents of this file is to implement the
**      functions the control execution of work item restores for
                                                                        Restore
**      Service  Library.
**
**      These functions are provided to allow:
**
** The following functions comprise restoral management:
**
**      RunWorkItemRestores()
**
**
** Compile-Time Options:
**              This section must list any compile time definitions
**              which will affect this header.
**
*****************************************************************************
                                                                          *****/



#define  _POSIX_SOURCE 1


/*
 * System headers.
 */

#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <values.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/ebutil.h>
#include <restore/REprogmsg.h>

/*
 * Local headers
 */

#include <RSLspexits.h>
#include <RSLremfd.h>
#include <EDMRESchedApi.h>
#include <EDMREHandleMgrApi.h>
#include <RSLinterns.h>
#include <RSLrbrmain.h>
#include <restore/EDMRESubmitApi.h>
#include <EDMREDrainApi.h>

#define STR_SURE(str) (str) ? str:""


/*
```



```
 *
 * RunWorkItemRestores
 * {
 * Set the number of drives being used for the life of the restore.
 * SetQuitFlag = FALSE
 * TrailRestoresLeft = { # of trail restores }
 * TrailRestoresRunning = 0;
 *
 *   while drive available.
 *     RunTrail -- Set drive concurrency for trail restore.
 *     TrailRestoresRunning++;
 *     while OKToRunWIForTrail
 *       StartWIRestore
 *
 *   while(1)
 *     if(QuitTest)
 *        Send WICancells
 *        SetQuitFlag = TRUE
 *     Select(from_pipe, timeout {5 seconds})
 *     for each WI that completes
 *       interperate return.
 *       Drain progress.
 *       Send final Progress for work item.
 *       if(WIfailed)
 *          if(OKtoReschedule && SetQuitFlag == FALSE)
 *             Add to TrailQueue
 *       if(TrailRestoreHasMoreWorkitems && SetQuitFlag == FALSE)
 *          while OKToRunWIForTrail
 *             StartWIRestore
 *       else -- RunNewTrailRestore
 *          EndTrailRestore(prevTrailQueue)
 *          TrailRestoresRunning--;
 *          TrailRestoresLeft--;
 *          while (drive available &&
 *                 SetQuitFlag == FALSE &&
 *                 TrailRestoresLeft)
 *
 *              RunTrail -- Set drive concurrency for trail restore.
 *              TrailRestoresRunning++;
 *              while (
 *                    OKToRunWIForTrail && SetQuitFlag == FALSE)
 *                StartWIRestore
 *              end while
 *           end while
 *        end else
 *    End for each WI completes
 *    if (((SetQuitFlag == TRUE) && (TrailRestoresRunning == 0))||
 *         TrailRestoresLeft == 0)
 *         return;  exit loop.
 *  end while(1)
 *
 * }
 *
 */

/* Stubs */
static int
InterpretWorkItemRestoreResults(wi_restore_results *results);

static int
test_fd(int fd);

static void
DebugLogFds(char *error_msg,
            fd_set *fds);
```

```
129   static int
130   DetermineGlobalDriveUse();

132   static int
133   test_fd_hup(int fd);

135   static int
136   FindTrailIDForWItem(int handle,
137                       int *TrailID,
138                       int *status);

140   static int
141   SendRunningWorkItemsQuit();

143   /* End Stubs */

145   static int
146   Select(int nfds,
147          fd_set *readfds,
148          fd_set *writefds,
149          fd_set *exceptfds,
150          struct timeval *timeout);


153   static int
154   HandleWorkItemRestoreResults(int FromFD,
155                                int *TrailID,
156                                wi_restore_results *results);
157   static int
158   RunWorkItemRestoresForTrail(const int TrailID,
159                               const int CountDrivesAvailable,
160                               boolean_ty (*CancelRestoreTest)(),
161                               boolean_ty *QuitFlag,
162                               int *CountDrivesInUse);

164   /*
165    * RunWorkItemRestores()
166    *
167    *  Runs a set of work item restores.
168    *
169    * Args:
170    *  SubmitObject
171    *  CancelRestoreTest().
172    *
173    * Returns: int 0 for success.
174    */
175   int
176   RunWorkItemRestores(int SubmitObject, boolean_ty (*CancelRestoreTest)(
                                                                        ))
177 1 {
178 1   boolean_ty QuitFlag = FALSE;  /* Has the user requested a quit.*/
179 1   boolean_ty SentQuit = FALSE;  /* Have we initiated the quit. */

181 1   int TrailRestoresRunning = 0;
                                /* The number of trail restores running. */
182 1   int TrailRestoresLeft;
                                 /* The number of trail restores left. */
183 1   int TrailRestoresTotal;
                                /* The number of trail restores total. */

185 1   int CountDrivesAvailable;     /* The count of drives available. */
186 1   int CountDrivesInUse = 0;     /* The count of drives in use. */

188 1   int temp_status;
189 1   int HighestActiveTrail = 0;
                           /* The trail queues are ordered from 1 to n. */
```



```
190 1                                 /* This is the highest trail running
                                                                        */


192 1   if(debugmode)
193 2   {
194 2     (void)rbe_user_error(0,
195 2                          "DEBUG: Running RunWorkItemRestores.");
196 1   }
197 1   /* GenerateTrailQueues()
198 1    * Buckets the work items into trail queues.
                                                  The trail queues are sorted
199 1    * in the order which the restores should run.
200 1    */

202 1   if(0 != GenerateTrailQueues(SubmitObject,
203 1                               &TrailRestoresTotal,
204 1                               &temp_status))
205 2   {
206 2     (void)rbe_user_error(0,
207 2                          "Internal error: Cannot generate trail
                                                  queues, cannot continue.");
208 2     return -1;
209 1   }

211 1   TrailRestoresLeft = TrailRestoresTotal;

213 1   CountDrivesAvailable = DetermineGlobalDriveUse(/*SubmitObject*/);

215 1   if(debugmode)
216 2   {
217 2     (void)rbe_user_error(0,
218 2                          "DEBUG: RunWorkItemRestores for %d trails.",
219 2                          TrailRestoresTotal);
220 1   }
221 1   /*
222 1    * This is the start up loop to get the initial work item
223 1    * restores started.
224 1    */

226 1   QuitFlag = CancelRestoreTest();

228 1   while((CountDrivesInUse < CountDrivesAvailable) &&
229 1         (HighestActiveTrail < TrailRestoresTotal) &&
230 1         (FALSE == QuitFlag))
231 2   {
232 2     int submitObjID = 0;
233 2     int submitelementID = 0;

235 2     HighestActiveTrail++;

237 2     /*
238 2      * Activate the Trail Queue.
                                  This allows the traill queues to be used to
239 2      * determine the work item restores to run.
240 2      */
241 2     if(0 != ActivateTrailQueue(HighestActiveTrail,
242 2                                1,
243 2                                &temp_status))
244 3     {
245 3       (void)rbe_user_error(0,
246 3                            "Internal error: Cannot activate trail
                                queues(1) for trailid %d, cannot continue.",
247 3                              HighestActiveTrail);
248 3       return -1;
249 2     }
```

```
251 2     /*
252 2      * This sets the number of drives and media access concurrency for
                                                                  trail restores.
253 2      * i.e. The count of running work item restores for this trail.
                                                                  Today this is one.
254 2      */
255 2     if(0 != SetTQDrivesAcquired(HighestActiveTrail, 1, &temp_status))
256 3     {
257 3       (void)rbe_user_error(0,
258 3                            "Internal error: Cannot set drive acquired(
                1) for trailid %d, cannot continue.", HighestActiveTrail);
259 3       return -1;
260 2     }

262 2     if (0 > (temp_status = RunWorkItemRestoresForTrail(
                                                          HighestActiveTrail,
263 2
                                                        CountDrivesAvailable,
264 2
                                                          CancelRestoreTest,
265 2                                                         &QuitFlag,
266 2
                                                       &CountDrivesInUse)))
267 3     {

269 3       /* RunWorkItemRestoresForTrail does its own error logging. */
270 3       return -1;
271 2     }

273 2     if(temp_status == 0)
274 3     {
275 3       (void)rbe_log_stats(0,
276 3                           "Trail %d restore had no work item to run(
                                                  1).", HighestActiveTrail);
277 3       /* more work may be need to recover from this error condition.
                                                                        */
278 2     }
279 2     if(temp_status > 0)
280 3     {
281 3       TrailRestoresRunning++;
282 2     }
283 1   } /* End while() initial startup loop */

285 1   while(1)
286 2   {
287 2     int HighestFd = 0;
288 2     fd_set WorkItemFromFds;
289 2     int retStatus;
290 2     struct timeval timeout = {5, 0};

292 2     if((QuitFlag) && (!SentQuit))
293 3     {
294 3       (void)rbe_log_stats(0,
295 3                           "Restore was quit by user. Quitting restore,
                                          this could take a while.");

297 3       SendRunningWorkItemsQuit();
298 3       SentQuit = TRUE;
299 2     }

301 2     if(0 != getFromSet(&WorkItemFromFds, &HighestFd, &retStatus))
302 3     {
303 3       (void)rbe_user_error(0,
304 3                            "Internal error: Cannot get auxproc result
                                          fds, cannot continue.");
```



```
306 3       return -1;
307 2     }
308 2   #if 0
309 2     if(debugmode)
310 3     {
311 3       DebugLogFds("The file descriptors to wait on are ",
312 3                   &WorkItemFromFds);
313 2     }
314 2   #endif
315 2     if(0 > (retStatus = Select(HighestFd + 1,
316 2                                &WorkItemFromFds,
317 2                                NULL, NULL,
318 2                                &timeout)))
319 3     { /* error */
320 3       (void)rbe_user_error(RBRECOVER_MKERR(errno),
321 3                            "Internal error: Cannot get auxproc result
                                                  fds, cannot continue.");

323 3       return -1;
324 2     }

326 2     else if (0 == retStatus)
327 3     { /* timed out */

329 3       QuitFlag = CancelRestoreTest();

331 2     }

333 2     else
334 3     { /* Available fds */

336 3       int ReadyFds = retStatus;
337 3       int FoundFds = 0;
338 3       int index;

340 3       if(debugmode)
341 4       {
342 4         DebugLogFds("The file descriptors ready to read are ",
343 4                     &WorkItemFromFds);
344 3       }

346 3       /*
347 3        * If there are available fds then we may want to
348 3        * schedule the next work item restore. We should
349 3        * check if the user initiated a quit.
350 3        */
351 3       QuitFlag = CancelRestoreTest();

353 3       for(index = 0;
354 3           (index < (HighestFd + 1)) && (FoundFds < ReadyFds);
355 3           index++)
356 4       {
357 4         int StartWorkItemforTrail = 0;
358 4         if(FD_ISSET(index, &WorkItemFromFds))
359 5         {
360 5           int TrailID;
361 5           int TrailAcquired;
362 5           wi_restore_results results;
363 5           FoundFds++;

365 5           memset(&results, 0, sizeof(wi_restore_results));

367 5           if(0 != HandleWorkItemRestoreResults(index,
368 5                                                &TrailID,
369 5                                                &results))
370 6           {
```

```
371 6             /* HandleWorkItemRestoreResults will do its own logging!
                                                                        */
372 6             return -1;
373 5           }
374 5           /*
375 5            * This is where we may want to retry the Work Item
376 5            * Based on if it passes or fails
377 5            */

379 5           CountDrivesInUse--;


382 5           if (0 > (StartWorkItemforTrail =
383 5                    RunWorkItemRestoresForTrail(TrailID,
384 5                                                CountDrivesAvailable,
385 5                                                CancelRestoreTest,
386 5                                                &QuitFlag,
387 5                                                &CountDrivesInUse)))
388 6           {
389 6             /* RunWorkItemRestoresForTrail does its own logging. */
390 6             return -1;
391 5           }

393 5           else if (StartWorkItemforTrail == 0)
394 5                /* 0 work items started aboved,
395 5                 * lets check to see if this is the last work item
                                                                left for
396 5                 * this trail
397 5                 */
398 6           {
399 6             int wiCount;

401 6             if(0 != GetRunningWI(TrailID, &wiCount, &temp_status))
402 7             {
403 7               (void)rbe_user_error(0,
404 7                                    "Internal error: Cannot determine
          number of running work items for trail, cannot continue.");

406 7               return -1;
407 6             }
408 6             if(debugmode)
409 7             {
410 7               (void)rbe_user_error(0,
411 7                                    "DEBUG: RunWorkItemRestores no
                                       more work items left for trailid %d ,
                                    but %d wiCount workitem still running.",
                                         TrailID, wiCount);
412 7             }
413 6             /* Testing for No work items left running or started
414 6              * For this trail.
415 6              */

418 6             if((0 == wiCount) && (0 == StartWorkItemforTrail))
419 7             {

421 7               TrailRestoresRunning--;
422 7               TrailRestoresLeft--;

424 7               if(0 != DeactivateTrailQueue(TrailID, &temp_status))
425 8               {
426 8                 (void)rbe_user_error(0,
427 8                                      "Internal error: Cannot
    deactivate trail queue for trailid %d, cannot continue.", TrailID);

429 8                 return -1;
430 7               }
```



```
432 7             /* This test is to determine,
                                          if a trail restore finished,
433 7              * there may be another not yet started trail,
                                                  If we have
434 7              * drive available the next trail restore will be
                                                       started.
435 7              */
436 7             if ((0 != TrailRestoresLeft) &&
437 7                 (HighestActiveTrail < TrailRestoresTotal) &&
438 7                 (CountDrivesInUse < CountDrivesAvailable))
439 8             {

441 8               HighestActiveTrail++;

443 8               if(0 != ActivateTrailQueue(HighestActiveTrail,
444 8                                          1,
445 8                                          &temp_status))
446 9               {
447 9                 (void)rbe_user_error(0,
448 9                                      "Internal error: Cannot
    activate trail queue(2) for trailid %d, cannot continue.",
449 9                                      HighestActiveTrail);
450 9                 return -1;
451 8               }

453 8               if(0 != SetTQDrivesAcquired(
                              HighestActiveTrail, 1, &temp_status))
454 9               {
455 9                 (void)rbe_user_error(0,
456 9                                      "Internal error: Cannot set
    drive acquired(2) for trailid %d, cannot continue.",
457 9                                      HighestActiveTrail);
458 9                 return -1;
459 8               }

461 8               if (0 > (temp_status = RunWorkItemRestoresForTrail(
                                                    HighestActiveTrail,
462 8                                                  CountDrivesAvailable,
463 8                                                    CancelRestoreTest,
464 8                                                          &QuitFlag,
465 8                                                   &CountDrivesInUse)))
466 9               {
467 9                 /* RunWorkItemRestoresForTrail does its own logging.
                                                                        */
468 9                 return -1;
469 8               }
470 8               if(temp_status == 0)
471 9               {
472 9                 /* If this Trail Had no work items we
473 9                  * Should attempt to run the next trails
474 9                  * work items here. This would be an internal
475 9                  * error if a trail queue had no work item
476 9                  * restores.
477 9                  */
478 9                 (void)rbe_log_stats(0,
479 9                                     "Internal error: Trail %d
    restore had no work item to run(1).", HighestActiveTrail);
480 9                 return -1;
481 8               }
482 8               if(temp_status > 0)
483 9               {
```

```
                    /* IF at least on work item was started for this
                                                                  trail,
                     * then we have started a new trail.
                     */
                    TrailRestoresRunning++;
                  }
                }

              }
            }
          }
        } /* end for() */

      } /* else Available fds */

      /*
       * Terminate the loop if either
       * ----------------------------
       * 1) Sent the work items the quit AND
       *    No Trail restores a running.
       * OR
       * 2) No more Trail restores are left.
       */

      if(((0 == TrailRestoresRunning) && (SentQuit)) ||
         (0 == TrailRestoresLeft))
      {
        break;
      }

    }  /* end while(1) */
    if((0 == TrailRestoresRunning) && (SentQuit))
    {
      (void)rbe_log_stats(0,
                          "Restore was quit by user.
                                                Work item restore quit.");
    }
    return 0;
  }
```



```

  /* Functions needed
  SendRunningWorkItemsQuit();
  InterpretWorkItemRestoreResults();
  */

  static int
  Select(int nfds,
         fd_set *readfds,
         fd_set *writefds,
         fd_set *exceptfds,
         struct timeval *timeout)
  {
    int retSelect;

    do
    {
      retSelect = select(nfds,
                         readfds,
                         writefds,
                         exceptfds,
                         timeout);

    } while ((-1 == retSelect) && (EINTR == errno));

    return retSelect;
  }
```

```
550   eperrno
551   InitiateWorkItemRestore(const int SubmitObjID,
552                           const int SubmitElemID)
553 1 {
554 1   struct auxproc AuxprocVitals;
555 1   eerrno_ty StartupAPResults = EXIT_FAILURE;
556 1   time_t StartTime;
557 1   int TempStatus;
558 1   char junk_executable[1024];
559 1   char **junk_argv;
560 1   char **AP_env = NULL;
561 1   int SOstatus;
562 1   char clientName[256] = "";
563 1   int clientPort;
564 1   int status;
565 1   time_t EndTime;
566 1   /* Lets see if there are any environment variables to set.
567 1    *  The restore of the output variables are ingnored.
568 1    */
569 1   if(E_SUCCESS != GetSOExecutionPhase(SubmitObjID,
570 1                                       junk_executable, 1024,
571 1                                       &junk_argv,
572 1                                       &AP_env,
573 1                                       &SOstatus))
574 2   {
575 2     (void)rbe_user_error(0,
576 2                          "Internal Error: Could not get environment
                                                                variables.");
577 2     return -1;
578 1   }

580 1   if (E_SUCCESS == GetSERcmdConnect(SubmitObjID,
581 1                                     SubmitElemID,
582 1                                     clientName, 256,
583 1                                     & clientPort,
584 1                                     &SOstatus))
585 2   {
586 2    StartupAPResults = StartupAuxprocess(0 /* XXX */,
587 2                                         &AuxprocVitals,
588 2                                         AP_env,
589 2                                         clientName,
590 2                                         clientPort);
591 1   }
592 1   else
593 2   {
594 2     (void)rbe_user_error(0,
595 2                          "Internal Error: Could not get Remote Client name
                                                                        &"
596 2                          "port to connect.");
597 2     return -1;
598 1   }

600 1   if(E_SUCCESS != StartupAPResults)
601 2   {
602 2     /* StartupAuxprocess does its own logging. */
603 2     return -1;
604 1   }

606 1   /*
607 1    * We need to close the bulk fd. This file descriptor
608 1    * is not being used any more. If we do not close it
609 1    * here we will have a file descriptor leak because
610 1    * we won't be able to determine what it was when the
611 1    * work item completes.
```



```
612 1    */

614 1   close(AuxprocVitals.xp_fd_bulk_to_x);

616 1   time(&StartTime);

618 1   if(0 != newHandleSet(AuxprocVitals.xp_fd_to_x,
619 1                        AuxprocVitals.xp_fd_from_x,
620 1                        AuxprocVitals.xp_fd_prog_from_x,
621 1                        SubmitObjID,
622 1                        SubmitElemID,
623 1                        AuxprocVitals.xp_pid,
624 1                        StartTime,
625 1                        &TempStatus))
626 2   {
627 2     (void)rbe_user_error(
                    0, "Internal Error: Could not register handle set.");
628 2     return -1;
629 1   }


632 1   if(0 > StartWorkItemRestore(rcp,
633 1                              &AuxprocVitals,
634 1                              SubmitObjID,
635 1                              SubmitElemID))
636 2   {
637 2     /*
638 2      * StartWorkItemRestore does logging if initialization fails
639 2      */
640 2      (void)rbe_user_error(
                  0, "Error in StartWorkItemRestore  SubmitObjID %d,"
641 2                           "SubmitElemID %d ",SubmitObjID,
                                                        SubmitElemID);
642 2      /*
643 2       * the following code kills auxproc when recx or xcpio do not
                                                                    start
644 2       * we do not want an auproc sitting around.
645 2       * if errors occur in deleteHandleSet or KillWorkItemRestore the
                                                            messages are
646 2       * logged in those calls, plus,
                                we already know there was an error and that
647 2       * is why we are doing this right now.
648 2       */
649 2      time(&EndTime);

651 2      deleteHandleSet(
 AuxprocVitals.xp_fd_from_x, EndTime, EP_RB_RECOVER_ALLFAIL, &status);

653 2      KillWorkItemRestore(
                        AuxprocVitals.xp_pid, AuxprocVitals.xp_fd_to_x);

655 2      return -1;
656 1   }

658 1   return 0;

660   } /* InitiateWorkItemRestore() */
```

```
663  /*
664   *       interperate return.
665   *       Drain progress.
666   *       Send final Progress for work item.
667   *       Delete the handle set.
668   */

670  static int
671  HandleWorkItemRestoreResults(int FromFD,
672                               int *TrailID,
673                               wi_restore_results *results)
674  1 {
675  1   int ret = 0;
676  1   int retries = 0;
677  1   int GetAuxprocResultsStatus;
678  1   int TempStatus;
679  1   int DrainResult;
680  1   int DrainedFD;
681  1   int wiCount;
682  1   int AuxProcPid;
683  1   int ToFD, getFromFD, ProgressFD;
684  1   time_t EndTime;
685  1   unsigned long jobstat;
686  1   int timeout = 3; /* Lets try 3 seconds */
687  1   boolean_ty fromFDHangUp = FALSE;

689  1   ToFD = getFromFD = ProgressFD = -1;

691  1   while(! (fromFDHangUp))
692  2   {
693  2     GetAuxprocResultsStatus = GetAuxprocResults(FromFD, results);

695  2     if(-1 == GetAuxprocResultsStatus)
696  3     {
697  3       /* GetAuxprocResults() does its own logging */

699  3       (void)rbe_user_error(0," Error in GetAuxprocResults");
700  3       return -1;
701  2     }
702  2     if(0 == GetAuxprocResultsStatus)
703  3     {
704  3         if(test_fd_hup(FromFD) == 1)
705  4           {
706  4             fromFDHangUp = TRUE;
707  3           }
708  2     }

710  2     /* The remote result are not always going to
711  2      * be set. For example if the remote command
712  2      * is not started correctly.
713  2      */

715  2     if(results->local_exit_set == TRUE)
716  3     {
717  3       break;
718  2     }
719  2     else
720  3     {
721  3       sleep(1);
722  3       test_fd(FromFD);
723  3       continue;
724  2     }
725  1   }
```



```
727  1   time(&EndTime);

729  1   if(0 != PushDrainRequest(FromFD, &TempStatus))
730  2   {
731  2     (void)rbe_user_error(0,
732  2                          "Internal error: Could not push drain
                                                request, cannot continue.");

734  2     return -1;
735  1   }
736  1   /* Lets give the progress thread a chance to drain keeping busy in
737  1    * the meanwhile.
738  1    */

740  1   if(0 != FindTrailQueueOfWI(FromFD, TrailID, &TempStatus))
741  2   {
742  2     (void)rbe_user_error(0,
743  2                          "Internal error: Could not find trail id for
                                    finished work item, cannot continue.");

745  2     return -1;
746  1   }

748  1   if(0 != DecrementRunningWI(*TrailID, &wiCount, &TempStatus))
749  2   {
750  2     (void)rbe_user_error(0,
751  2                          "Internal error: Could not decrement running
                                   work items for trail, cannot continue.");

753  2     return -1;
754  1   }

756  1   if(0 != getPID(FromFD, &AuxProcPid, &TempStatus))
757  2   {
758  2     (void)rbe_user_error(0,
759  2                          "Internal error: Could not get auxproc pid
                                          for work item, cannot continue.");

761  2     return -1;
762  1   }

764  1   if(0 != getHandleSet(
                  FromFD, &ToFD, &getFromFD, &ProgressFD, &TempStatus))
765  2   {
766  2     (void)rbe_user_error(0,
767  2                          "Internal error: Could not get auxproc file
                                descriptors for work item, cannot continue.");

769  2     return -1;
770  1   }
771  1   if(FromFD != getFromFD)
772  2   {
773  2     (void)rbe_user_error(0,
774  2                          "Internal error:  mismatch on from file
                                descriptors for work item, cannot continue.");
775  2     return -1;
776  1   }

778  1   while (0 != (ret = PopDrainResult(timeout,
779  1                                     &DrainResult,
780  1                                     &DrainedFD,
781  1                                     &TempStatus)) && retries < 3)
782  1     retries++;

784  1   if (ret != 0)
785  2   {
```

```
786 2     (void)rbe_user_error(0,
787 2                          "Internal error: Could not pop drain results,
                                                        cannot continue.");


789 2     return -1;
790 1   }

792 1   /*
793 1    *       Send final Progress for work item. XXX
794 1    */

796 1   /* Translate the local and remote error statuses
797 1    * to an eperrno value:
798 1    */

800 1   if(0 != results->local_exit_status)    /* use local error, if any */
801 1     switch (results->local_exit_status)
802 2     {
803 2     case XG_EXIT_ALLFAIL:
804 2       jobstat = EP_RB_RECOVER_ALLFAIL;
805 2       break;
806 2     case XG_EXIT_MANYFAIL:
807 2       jobstat = EP_RB_RECOVER_MANYFAIL;
808 2       break;
809 2     case XG_EXIT_FEWFAIL:
810 2       jobstat = EP_RB_RECOVER_FEWFAIL;
811 2       break;
812 2     case SPEXIT_REMOTE_STDERR_PROTOCOL:
813 2     case SPEXIT_REMOTE_STDERR_FAIL:
814 2       jobstat = EP_RB_RECOVER_CLIENT_STDERR_FAIL;
815 2       break;
816 2     case XG_EXIT_STOPPED:         /* treat like signal */
817 2     default:      /* check for signal termination vs all generic
                                                          failures */
818 2       if ( XG_EXIT_SIGBASE < results->local_exit_status
819 2         || XG_EXIT_STOPPED == results->local_exit_status )
820 3       { /* killed by signal or stopped: separate error for sigpipe */
821 3         if (XG_EXIT_SIGBASE + SIGPIPE == results->local_exit_status)
822 3           jobstat = EP_RB_RECOVER_SERVER_SIGPIPE;
823 3         else
824 3           jobstat = EP_RB_RECOVER_SERVER_SIGNAL;
825 2       }
826 2       else
827 3       { /* generic server failure, unless client failed too */
828 3         jobstat = EP_RB_RECOVER_SERVERFAIL;
829 3         if (0 != results->remote_exit_status)
830 3           jobstat = EP_RB_RECOVER_BOTHFAIL;
831 2       }
832 1     }
833 1   else if(0 != results->remote_exit_status)
834 1     jobstat = EP_RB_RECOVER_CLIENTFAIL;
835 1   else
836 1     jobstat = E_SUCCESS;

838 1   if( (0 != results->remote_exit_status) ||
839 1       (0 != results->local_exit_status ) )
840 2   {
841 2    int  status=0;
842 2    int  rc=0;
843 2    char *templateName=NULL;
844 2    char *winame=NULL;
845 2    char *trailsetName=NULL;

847 2    rc = getHandleSetInformation(FromFD,
848 2                                 &templateName,
849 2                                 &winame,
```



```
850 2                                &trailsetName,
851 2                                &status);
852 2    rbe_log_stats(0, "Restore Failure of "\
853 2                  "top level object: %s, template %s.",
854 2                  STR_SURE(winame),
855 2                  STR_SURE(templateName));
856 2    free(templateName);
857 2    free(winame);
858 2    free(trailsetName);
859 1   }

861 1   if( 0 != deleteHandleSet(FromFD, EndTime, jobstat, &TempStatus))
862 2   {
863 2         (void)rbe_user_error(0,
864 2                              "Internal error: Could not delete Handle
                                                    Set, cannot continue.");


866 2     return -1;
867 1   }

869 1   if(0 != KillWorkItemRestore(AuxProcPid,
870 1                               -1 /* Hack this arg is not needed yet
                                                              cmd_to */))

871 2   {
872 2     (void)rbe_user_error(0,
873 2                          "Internal error: Could not kill finished
                                                  auxproc, cannot continue.");
874 2     return -1;
875 1   }

877 1   close(ToFD);
878 1   close(FromFD);
879 1   close(ProgressFD);

881 1   if(debugmode)
882 2   {
883 2     (void)rbe_user_error(0,
884 2                          "DEBUG: HandleWorkItemRestoreResults Auxproc(
            PID %d) just finished for trailid %d work items left = %d.",
885 2                          AuxProcPid,
886 2                          *TrailID,
887 2                          wiCount);

889 2     (void)rbe_user_error(0,
890 2                          "DEBUG: HandleWorkItemRestoreResults Auxproc(
           PID %d) results are local: %d: setP:%s remote: %d set:%s.",
891 2                          AuxProcPid,
892 2                          results -> local_exit_status,
893 2                          (
                            results -> local_exit_set) ? "TRUE": "FALSE" ,
894 2                          results -> remote_exit_status,
895 2                          (
                           results -> remote_exit_set) ? "TRUE": "FALSE");


898 1   }
899 1   return 0;
900   } /* End HandleWorkItemRestoreResults() */
```

```
 904  /*
 905   * RunWorkItemRestoresForTrail()
 906   *
 907   * Description
 908   *  This function starts all the work item for the
 909   *  trail. For no this is set to one but concurrency
 910   *  will can be supportted.
 911   *
 912   * Args:
 913   *  (I) TrailID -- The id for this trail.
 914   *  (I) CountDrivesAvailable -- the total drives available to restore.
 915   *  (O) QuitFlag -- indicatate whether the user has quit the restore.
 916   *  (O) CountDrivesInUse -- The count of trails in use by restore.
 917   *
 918   * Return int
 919   *  if -1 then an error has occurred.
 920   *  if 0 or greater then the number of trail restores started will be
 921   *  returned.
 922   *
 923   */
 924  static int
 925  RunWorkItemRestoresForTrail(const int TrailID,
 926                              const int CountDrivesAvailable,
 927                              boolean_ty (*CancelRestoreTest)(),
 928                              boolean_ty *QuitFlag,
 929                              int *CountDrivesInUse)
 930 1{
 931 1  int DriveAcquiredForTrail;
 932 1  int DriveConcurencyForTrail;
 933 1  int submitObjID;
 934 1  int submitelementID;
 935 1  int popResults = 0;
 936 1  int temp_status;
 937 1  int CountOfWorkItemRestoreStarted = 0;
 938 1  int wiCount;

 940 1  while(1)
 941 2  {



 944 2    (*CountDrivesInUse)++;

 946 2    if((0 != (popResults = PopWIFromTrailQueue(TrailID,
 947 2                                               &submitObjID,
 948 2                                               &submitelementID,
 949 2                                               &temp_status))) &&
 950 2       (SCHED_NO_MORE_JOBS != temp_status))
 951 3    {
 952 3      (void)rbe_user_error(0,
 953 3                           "Internal error: Cannot pop work item off
                                      trail queue, cannot continue.");
 954 3      return -1;
 955 2    }

 957 2    if((-1 == popResults) && (SCHED_NO_MORE_JOBS == temp_status))
 958 3    {
 959 3      return CountOfWorkItemRestoreStarted;
 960 2    }

 962 2    temp_status = InitiateWorkItemRestore(
                                        submitObjID, submitelementID);

 964 2    if(temp_status != 0)
```



```
 965 3    {
 966 3      /* InitiateWorkItemRestore() does its own logging */
 967 3      (void)rbe_user_error(0, "Error in InitiateWorkItemRestore,"
 968 3           "submitObjID  %d, submitelementID %d", submitObjID,
                                                          submitelementID);
 969 3      return -1;
 970 2    }

 972 2    if(0 != IncrementRunningWI(TrailID, &wiCount, &temp_status))
 973 3    {
 974 3      (void)rbe_user_error(0,
 975 3                           "Internal error: Could not increment
                         running work items for trail, cannot continue.");
 976 3      return -1;
 977 2    }

 979 2    CountOfWorkItemRestoreStarted++;

 981 2    if(0 != GetTQDrivesAcquired(TrailID,
 982 2                                &DriveAcquiredForTrail,
 983 2                                &temp_status))
 984 3    {
 985 3      (void)rbe_user_error(0,
 986 3                           "Internal error: Cannot get drives
                                        acquired, cannot continue.");

 988 3      return -1;
 989 2    }

 991 2    if(0 != GetTQDriveConcurrency(TrailID,
 992 2                                  &DriveConcurencyForTrail,
 993 2                                  &temp_status))
 994 3    {
 995 3      (void)rbe_user_error(0,
 996 3                           "Internal error: Cannot get drive
                                      concurrency, cannot continue.");
 997 3      return -1;
 998 2    }

1000 2    *QuitFlag = CancelRestoreTest();

1002 2    if((DriveAcquiredForTrail < DriveConcurencyForTrail) &&
1003 2       ((*CountDrivesInUse) < CountDrivesAvailable) &&
1004 2       (FALSE == *QuitFlag))
1005 3    {
1006 3      continue;
1007 2    }
1008 2    else
1009 3    {
1010 3      break;
1011 2    }
1012 1  }
1013 1  return CountOfWorkItemRestoreStarted;
1014  } /* RunWorkItemRestoresForTrail() */
```

```
1018    /* Stub */
1019    static int DetermineGlobalDriveUse()
1020  1 {
1021  1 /* Limiting to MAXINT === not limiting... Need resource management
1022  1  * to do this properly.
                        NOTE: This should now work like eb_dc_restore does.
1023  1  */
1024  1   return MAXINT;
1025    }
```



```
1028    static int
1029    SendRunningWorkItemsQuit()
1030  1 {
1031  1   int *APlist;
1032  1   int count;
1033  1   int status;
1034  1   int index;

1036  1   if(0 != getPIDList(&count, &APlist, &status))
1037  2   {
1038  2     (void)rbe_user_error(0,
1039  2                          "Internal error: Cannot get auxproc pid list,
                                                                cannot continue.");
1040  2     return -1;
1041  1   }

1043  1   for(index = 0; index < count; index++)
1044  2   {
1045  2     QuitWorkItemRestore(APlist[index]);
1046  1   }
1047  1   return 0;
1048    }
```

```
1050    /*
1051     * Stub this out for now.
1052     */
1053    static int
1054    InterpretWorkItemRestoreResults(wi_restore_results *results)
1055  1 {
1056  1   return 0;
1057    }
```



```
1059    static void
1060    DebugLogFds(char *error_msg,
1061                fd_set *fds)
1062  1 {
1063  1   int index, fd_count = 0;
1064  1   char buffer[4096];
1065  1   char *bufptr = (char*)buffer;


1068  1   for(index=0;
1069  1       index < 1024;
1070  1       index++)
1071  2   {
1072  2     if( FD_ISSET(index, fds))
1073  3     {
1074  3       int size = 0;
1075  3       size = sprintf(bufptr,"%d,",index);
1076  3       bufptr += size;
1077  3       fd_count++;
1078  2     }
1079  1   }
1080  1   rbe_log_stats(0,"%s fd_count:: %d :: {%s}\n",
1081  1                 error_msg, fd_count, buffer);
1082    }
```

```
1085    static int
1086    test_fd(int fd)
1087  1 {
1088  1   fd_set read_fd;
1089  1   int ret_select;
1090  1   struct timeval timeout = {0, 0};

1092  1   FD_ZERO(&read_fd);

1094  1   FD_SET(fd, &read_fd);

1096  1   do
1097  2   {
1098  2     ret_select = select(fd + 1, &read_fd, NULL, NULL,  &timeout);

1100  1   } while((-1 == ret_select) && (EINTR == errno));

1102  1   return ret_select;

1104    }
```



```
1106    /*
1107     * test_fd_hup()
1108     *
1109     * Description: Test the supplied file descriptor to see if
1110     *   it has had the hang up condition.
1111     *
1112     * Args:
1113     *  Input fd -- the file descriptor to check for the hang up condition.
1114     *
1115     * Returns:
1116     *  1 for HUP event received on fd.
1117     *  0 No HUP event received on fd.
1118     *  -1 errno set.
1119     *
1120     */
1121    static int
1122    test_fd_hup(int fd)
1123  1 {
1124  1   struct pollfd fds;
1125  1   int ret_poll;

1127  1   if(fd < 0)
1128  2   {
1129  2     errno = EINVAL;
1130  2     return -1;
1131  1   }

1133  1   fds.fd = fd;
1134  1   fds.events = POLLIN;
1135  1   fds.revents = 0;  /* initialize */

1137  1   do
1138  2   {
1139  2     ret_poll =  poll(&fds, 1, 0);

1141  1   } while((-1 == ret_poll) && (EINVAL == errno));

1143  1   if(-1 == ret_poll)
1144  2   {
1145  2     return -1;
1146  1   }

1148  1   if(POLLHUP & fds.revents)
1149  2   {
1150  2     return 1;
1151  1   }
1152  1   else
1153  2   {
1154  2     return 0;
1155  1   }

1157    } /* end test_fd_hup() */
```

```
/*****************************************************************************
                                                                          ******
**
** File Name:   RSLauxmgr.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
** -----------
**      The intent of the contents of this file is to implement the
**      functions the control execution of work item restores Or the
**      running of auxproc.
**
**
**      Library.
**
**      These functions are provided to allow:
**          - The pipe, fork, duping closing and exec of auxproc.
**          - The starting of work item restores.
**          - The quitting of work item restores.
**          - The getting results of work item restores.
**
**
** The following functions comprise restoral management:
**
**
**
**
**
** Compile-Time Options:
**              This section must list any compile time definitions
**              which will affect this header.
**
*****************************************************************************
                                                                          *****/



/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */

#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/eb_normalize.h>
#include <ebutil/ebutil.h>
#include <ebreport/ebvl.h>
```



```
/*
 * Local headers
 */

#include <RSLinterns.h>
#include <RSLrbrmain.h>
#include <restore/EDMRESubmitApi.h>
#include <EDMfork.h>
#include <RSLauxSupp.h>

#define SUBMIT_FIELD_MAX 2048

extern int putenv(const char *string);

extern char *strsignal(int sig);

extern int ChildDone(int child_pid, int *child_result);

static void
reset_recovery_privileges(struct recover_context *rcx,
                          int changed);

static void
set_recovery_privileges(struct recover_context *rcx,
                        int *changed);

static char *
generate_rcmdpath(int SubmitObjectID, int SubmitElemID);

/*
 * StartupAuxprocess()
 *
 * Description:
 *  This function pipe, fork,  & exec auxproc. After which it tests
 *  its the just started auxproc with the Ping command.
                                                          Some additional
 *  processing is it closes the fds that are not associated with
                                                                auxproc
 *  but are inherited from the parent (
                                          restore engine). If any environment
 *  variable need to be set for auxproc, they are set.
 *
 * Args:
 * (Inp) debugmode -- int 1 for debug and 0 for no debug.
 * (Out) struct auxproc *xp -- preallocated structure to return vital
                                                                      info
 *                          for auxproc.
 * (char) **auxproc_envp -- environment variables to be appended to
                                                            auxproc's
 *                          environment.
                                    The rest of the environment is
 *                          inheritted from auxproc.
 *                          Format is "ENV=value\0"
 * char  *socketClientNm  -- used for the client initiated restore,
                                                          this is the
 *                          client name
 * int  clientSocketPort -- Used to identify the port number on which
                                                                   to
 *                          contact the client
 * Notes:
 *     auxproc_envp can be NULL indicating no environment to append to
                                                                auxproc.
 *     This is a char ** which last char * should be NULL.
 *
 * Inheritted from:
```

```
 122    *  static eerrno_ty  setup_aux_processes(struct recover_context *rcx)
 123    */

 125    eerrno_ty
 126    StartupAuxprocess(int          debugmode,
 127                      struct       auxproc *xp,
 128                      char         **auxproc_envp,
 129                      char   *socketClientNm,
 130                      int    clientSocketPort)
 131  1 {

 133  1   /*
 134  1    * NOTES::
 135  1    * 1) Do I really want to be reliant on the recover_context struct.
 136  1    * 2) I need to fork() exec() auxproc.
 137  1    * 3) Is fork1 really going to work.
 138  1    */



 141  1 #define RFD      0       /* in a pipe, fd 0 is the read descriptor */
 142  1 #define WFD      1       /* and fd 1 is the write descriptor */
 143  1     int save_errno;
 144  1     int fd;
 145  1     char *resultsbuf = NULL;
 146  1     char *Auxproc_pathname = AUXPROCPATHNAME;
 147  1     char *Auxproc_executable = AUXPROCNAME;

 149  1     int ping_status;
 150  1     int auxproc_index = 0; /* alton Remove */
 151  1     int cmd_pipe_to[2];
 152  1     int cmd_pipe_from[2];
 153  1     int bulk_pipe_to[2];
 154  1     int prog_pipe_from[2];

 156  1     if (pipe(cmd_pipe_to) == -1 ||
 157  1         pipe(cmd_pipe_from) == -1 ||
 158  1         pipe(bulk_pipe_to) == -1 ||
 159  1         pipe(prog_pipe_from) == -1)
 160  2     {
 161  2         save_errno = errno;
 162  2         rbe_log_stats(RBRECOVER_MKERR(errno), "pipe() failed");
 163  2         return(RBRECOVER_MKERR(save_errno));

 165  1     }
 166  1     /* This below appends environment variables to
 167  1      * the evironment that auxproc inherits from the
 168  1      * restore engine.
 169  1      */

 171  1     if(NULL != auxproc_envp)
 172  2       {
 173  2         int index;
 174  2         for(index=0; NULL != auxproc_envp[index]; index++)
 175  3           {
 176  3             if(0 != putenv(auxproc_envp[index]))
 177  4               {
 178  4                 rbe_log_stats(RBRECOVER_MKERR(errno),
 179  4                               "Unable to set auxproc environment %s,
                                                    for auxproc PID %d.",
 180  4                               auxproc_envp[index], getpid());

 182  4                 _exit(1); /* We are the child */
 183  3               }
 184  2           }

 186  1       }
```



```
 188  1     switch (xp->xp_pid = EDMfork1())
 189  2     {
 190  2         case -1: /* Error */
 191  2            save_errno = errno;
 192  2            rbe_log_stats(RBRECOVER_MKERR(errno), "fork() failed");
 193  2            return(RBRECOVER_MKERR(save_errno));

 195  2          case 0: /* child */


 198  3            {
 199  3            char procnum_str[32];
 200  3            char r_fd_str[32];
 201  3            char w_fd_str[32];
 202  3            char r_bulk_fd_str[32];
 203  3            char w_prog_fd_str[32];
 204  3            char dbgmodestr[32];
 205  3           char socket_host_str[256];
                                          /* Not sure what this should be */
 206  3           char socket_port_str[32];
 207  3            int ret_exec = 0;
 208  3           socket_port_str[0] = '\0';

 210  3            (void)sprintf(procnum_str, "%d", auxproc_index);
 211  3            (void)sprintf(r_fd_str, "%d", cmd_pipe_to[RFD]);
 212  3            (void)sprintf(w_fd_str, "%d", cmd_pipe_from[WFD]);
 213  3            (void)sprintf(r_bulk_fd_str, "%d", bulk_pipe_to[RFD]);
 214  3            (void)sprintf(w_prog_fd_str, "%d", prog_pipe_from[WFD]);
 215  3            (void)sprintf(dbgmodestr, "%d", debugmode);
 216  3            if (NULL != socketClientNm)
 217  4              {
 218  4                (void)sprintf(socket_host_str, "%s", socketClientNm);
 219  4                (void)sprintf(socket_port_str, "%d", clientSocketPort);
 220  3              }
 221  3            else
 222  4              {
 223  4               socket_host_str[0] = 0;
 224  4               socket_port_str[0] = 0;

 226  3              }
 227  3            ret_exec = execlp(
                              Auxproc_pathname,          /* prog to execute */
 228  3                                Auxproc_executable,
                                                         /* argv 0 */
 229  3                                procnum_str,
 230  3                                r_fd_str,
 231  3                                w_fd_str,
 232  3                                r_bulk_fd_str,
 233  3                                w_prog_fd_str,
 234  3                                dbgmodestr,
 235  3                                socket_host_str,
 236  3                                socket_port_str,
 237  3                                (char *)0);

 239  3            rbe_log_stats(RBRECOVER_MKERR(errno),
 240  3                          "Unable to exec %s, for %s PID %d.",
 241  3                          AUXPROCNAME,
 242  3                          AUXPROCNAME,
 243  3                          getpid());


 246  3            _exit (1);
 247  2            }

 249  2     default: /* parent */
```

```
251  2        /*
252  2         * The parent has no need for the fd
253  2         * used to write *to* the parent, nor
254  2         * the fds used to read *from* the parent.
255  2         * If these are not close EPIPE and SIGPIPE
256  2         * may be missed by the writer when the
257  2         * intended reader dies.
258  2         */

260  2        (void)close(cmd_pipe_to[RFD]);
261  2        (void)close(bulk_pipe_to[RFD]);
262  2        (void)close(cmd_pipe_from[WFD]);
263  2        (void)close(prog_pipe_from[WFD]);

265  2        /*
266  2         * but does want to save the other fds
267  2         */

269  2        xp->xp_fd_from_x     = cmd_pipe_from[RFD];
270  2        xp->xp_fd_to_x       = cmd_pipe_to[WFD];
271  2        xp->xp_fd_bulk_to_x = bulk_pipe_to[WFD];
272  2        xp->xp_fd_prog_from_x = prog_pipe_from[RFD];
273  2        /*
274  2         * In debugmode, exec the separate-process
275  2         * version of the auxprocs (ptrace issue)
276  2         */

278  2        if (debugmode)
279  3        {
280  3           auxcmdpacket(xp->xp_fd_to_x, 'X', 0, "");
281  2        }
282  1      }
283  1  #define PING_TEST_STR "abc"
284  1  #define PING_TEST_STR_SIZE 4 /* include the '\0' */

286  1      fd = xp->xp_fd_to_x;
287  1      auxcmdpacket(fd, 'P', PING_TEST_STR_SIZE, PING_TEST_STR);

289  1      fd = xp-> xp_fd_from_x;

291  1      ping_status = auxresults(fd, 'P', 0, &resultsbuf);

293  1      if ((-1 == ping_status) ||
294  1           (NULL == resultsbuf) ||
295  1           (strcmp(resultsbuf, PING_TEST_STR) != 0))
296  2      {
297  2        rec_api_log_csm(SUB_CSM_NO_PING_AUXPROC, NULL);
298  2        rbe_log_stats(0, "%s ping start-up",
299  2                      AUXPROCNAME);
300  2        return(EP_RB_RECOVER_AUXPROC_DIED);
301  1      }

303  1      free(resultsbuf);

305  1      return(E_SUCCESS);

307  1  #undef PING_TEST_STR
308  1  #undef PING_TEST_STR_SIZE
309  1  #undef RFD
310  1  #undef WFD
311  1  #undef MAX_FD
312     }        /* end of setup_aux_process() */
```



```
315  #define MAX_SUBMIT_FIELD 2048


318  /*******************************************************
319   * start_cpiogen()
320   *
321   *  This function initiates a work item restore. It first determines
322   *  the size of the restore command to send to auxproc, malloc's the
323   *  memory and creates the restore command and sends it to auxproc.
324   *  Auxproc will start the rcmd (if necessary) and start xcpiogen.
325   *  Auxproc will send the initialization reply which is read by this
326   *  function. The results are for initialization.
327   *
328   * Args:
329   *  (I) rcx -- Recover Context struct (Limited use)
330   *  (I) xp -- auxproc struct pointer.
331   *  (I) submitObjectID -- idententifies what and how to run restore.
332   *  (I) submitElemID --  idententifies what and how to run witem
                                                                  restore.
333   *  (I) auxprocnum -- auxproc number may become obsolete.
334   *  (I) *rcmdinfo[4] -- the rcmdinfo must be included.
335   *  (O) results -- Of the work item restore initialization.
336   *  (O) err_str -- Of work item initialization failures.
337   *
338   * Returns:
339   *  xcpiogen's pid for success, -1 for failure.
340   *
341   * NOTES::
342   *  rcx -- The recover context structure is carefully used below.
343   *  The rcx structure should be used only to get the global values
344   *  like the xcpiogen exectutable name and the config structure.
345   *
346   *  Submit Object should be used to determine user id,
347   *  admin priveledges, and other values that would not vary
348   *  for a potentially multi work item restore.
349   *
350   *  Submit Element  should be used to determine anything that
351   *  could be potentially unique for a work item restore.
352   *
353   *  rcmdinfo is the command line for the remote command.
354   *******************************************************/
355  int
356  start_cpiogen(struct recover_context *rcx,
357                struct auxproc *xp,
358                int submitObjectID,
359                int submitElemID,
360                int auxprocnum,
361                char *rcmdinfo[4],
362                rcmd_pkt0_info *results,
363                char **err_str)
364  1  {
365  1      char *auxproc_databuf;
366  1      size_t data_len = 0;
367  1      char *p;
368  1      int i;
369  1      int resfd;

371  1      /* For Initialization results */
372  1      int pid = -1;
373  1      rcmd_pkt0_info *pkt0p;
374  1      rcmd_pkt0_info pkt0p_buffer;
375  1      char *resultsbufptr = NULL;
376  1      char *errstr = "";
377  1      struct mark_summary submit_summary;
```

```
379 1     /* Args for xcpiogen */
380 1     char *xcpiogen_argv0;
381 1     int xcpiogen_argc;
382 1     char *submit_file_flag = "-S";
383 1     char *outputfd_fmt = "-f%d";
384 1     char *progress_report_flag = "-p";
385 1     u_hyper total_bytes;                    /* From the summary */
386 1     char *total_bytes_flag = "-B";
387 1     char *total_bytes_stringP;
388 1     char *bufsizeP = NULL;
389 1     char bufsize_buffer[20];

391 1     RBC_WORKGROUP *pg;
392 1     RBC_WORKITEM *pi;

394 1     /* temporary variable for the submit object / element fields. */
395 1     char temp_effective_uidname[MAX_SUBMIT_FIELD];
396 1     char temp_socket_host[MAX_SUBMIT_FIELD];
397 1     char temp_workitem_name[MAX_SUBMIT_FIELD];
398 1     char temp_submit_file[MAX_SUBMIT_FIELD];
399 1     int  temp_socket_port;
400 1     int GetSEStatus = 0;
401 1     int GetSOStatus = 0;

403 1     if( GetSEWorkItemName(submitObjectID, submitElemID,
404 1                           temp_workitem_name, MAX_SUBMIT_FIELD,
405 1                           &GetSEStatus) != 0)
406 2     {
407 2       rbe_log_stats(0, "Unable to get work item name.");
408 2       return -1;
409 1     }

411 1     if( GetSOEffectiveUserName(submitObjectID,
412 1                                temp_effective_uidname,
413 1                                MAX_SUBMIT_FIELD, &GetSOStatus) != 0)
414 2     {
415 2       rbe_log_stats(0, "Unable to get user name.");
416 2       return -1;
417 1     }


420 1     if( GetSERcmdConnect(submitObjectID, submitElemID,
421 1                          temp_socket_host,  MAX_SUBMIT_FIELD,
422 1                          &temp_socket_port, &GetSEStatus) != 0)
423 2     {
424 2       rbe_log_stats(0, "Unable to get socket host/port pair.");
425 2       return -1;
426 1     }

428 1     if( GetSESubmitFile(submitObjectID, submitElemID,
429 1                         temp_submit_file,  MAX_SUBMIT_FIELD,
430 1                         &GetSEStatus) != 0)
431 2     {
432 2       rbe_log_stats(0, "Unable to get submit file.");
433 2       return -1;
434 1     }


437 1     /*
438 1      * +1 for '\0' characters
439 1      */

441 1     data_len += strlen(rcmdinfo[0]) + 1;          /* rcmd-hostname */
442 1     data_len += strlen(rcmdinfo[1]) + 1;          /* rcmd-locuser */
443 1     data_len += strlen(rcmdinfo[2]) + 1;          /* rcmd-remuser */
```



```
444 1     data_len += strlen(rcmdinfo[3]) + 1;          /* rcmd-cmd */

446 1     data_len += strlen(temp_effective_uidname) + 1;

448 1     data_len += sizeof (int);                              /* fd info */
449 1     data_len += sizeof (int);                              /* flags */
450 1     data_len += strlen(rcx -> rc_cpiogen_executable) + 1;
                                                        /* xcpiogen-cmd */
451 1     data_len += sizeof (int);
                                                       /* xcpiogen-argc */


453 1     /*
454 1      * xcpiogen arguments.  Pass traditional argv[0]
455 1      * plus the output file descriptor number,
456 1      * plus -p (progress report mode) and -B with its
457 1      * argument (total bytes to be processed).
458 1      */

460 1     xcpiogen_argc = 7;

462 1     xcpiogen_argv0 = strrchr(rcx -> rc_cpiogen_executable, '/');
463 1     if (NULL != xcpiogen_argv0)
464 2     {
465 2          ++xcpiogen_argv0;
466 1     }
467 1     else
468 2     {
469 2          xcpiogen_argv0 = rcx -> rc_cpiogen_executable;
470 1     }
471 1     data_len += strlen(xcpiogen_argv0) + 1;
472 1     data_len += strlen(submit_file_flag) + 1;
                                                    /* xcpiogen argv[1] */
473 1     data_len += strlen(temp_submit_file) + 1;
                                                    /* xcpiogen argv[2] */
474 1     data_len += strlen(outputfd_fmt) + 1;
                                                    /* xcpiogen argv[3] */
475 1     data_len += strlen(progress_report_flag) + 1;
                                                    /* xcpiogen argv[4] */


477 1     /*
478 1      * Get the current total number of bytes to be processed from
479 1      * the mark summary u_hyper so that we can then convert it to
480 1      * a decimal string to be passed to xcpiogen().
481 1      */

483 1     if(0 != GetSEMarkedSummary(submitObjectID,
484 1                                submitElemID,
485 1                                &submit_summary,
486 1                                &GetSEStatus))
487 2     {
488 2       /* This is not a critical error. This may cause progress
489 2        * reporting problems!
490 2        */

492 2       rbe_log_stats(0, "Unable to set submit size for xcpiogen.");
493 2       total_bytes = ul_to_uh(0);
494 1     }
495 1     else
496 2     {
497 2       total_bytes = submit_summary.len_mkd_files;
498 1     }

500 1     total_bytes_stringP = u_hyper_to_decimal(total_bytes);

502 1     /*
503 1      * Add the size of the "total bytes" flag and value string
```

```
504 1       */
506 1      data_len += strlen(total_bytes_flag) + 1;
                                          /* xcpiogen argv[5] */
507 1      data_len += strlen(total_bytes_stringP) + 1;
                                          /* xcpiogen argv[6] */

509 1      /*
510 1       * Add size for db API socket info
511 1       */

513 1      data_len += sizeof (int);                  /* socket port # */
514 1      data_len += strlen(temp_socket_host) + 1;

516 1      /*
517 1       * locate work item in config info & get length of filespec
518 1       * %%% changed break in inner loop to 'goto' to resume with
519 1       *     found item.
520 1       */

522 1      for (pi = NULL, pg = rcx->rc_config->pgrouplist /* OK */;
523 1           NULL != pg;
524 1           pg = pg->next)
525 2      {
526 2           for (pi = pg->pwilist; NULL != pi; pi = pi->next)
527 3           {
528 3                if (0 == strcmp(pi->name, temp_workitem_name))
529 4                {
530 4                     goto stopsearch;      /* %%% exit both loops */
531 3                }
532 2           }
533 1      }

535 1  stopsearch:
536 1      if (pi != NULL)
537 2      {
538 2           data_len += strlen(pi->list)+1;
539 1      }
540 1      else
541 2      {
542 2           data_len += 1;
543 1      }

545 1      if (NULL != pi && DEFAULT_EB_BUFSIZE !=
                                             pi->recover_server_bufsize)
546 2      {
547 2           sprintf(bufsize_buffer, "-R%d", pi->recover_server_bufsize);
548 2           bufsizeP = bufsize_buffer;
549 2           data_len += strlen(bufsizeP) + 1;
550 2           xcpiogen_argc++;         /* one more arg to xcpiogen */
551 1      }

553 1      data_len += strlen(temp_workitem_name) + 1;

555 1      /*
556 1       * Allocate memory to hold all this gunk we need to
557 1       * shove towards our auxilliary process.
558 1       */

560 1      auxproc_databuf = sm_fmalloc((unsigned)data_len);

562 1      /*
563 1       * Fill in the gunk.  First, the rcmd info for the rsh part.
564 1       */

566 1      p = auxproc_databuf;
```



```
567 1      for (i = 0; i < 4; i++)
568 2      {
569 2           (void)strcpy(p, rcmdinfo[i]);
570 2           p += strlen(p)+1;
571 1      }

573 1      /*
574 1       * The human username which the remote should operate as
575 1       */

577 1      (void)strcpy(p, temp_effective_uidname);
578 1      p += strlen(p)+1;

580 1      /*
581 1       * The xcpiogen-cmd-fd-info, which tells auxproc which
582 1       * arg has the sprintf format string to insert the
583 1       * actual output fd number. Remember auxproc sets up
584 1       * the connections between xcpiogen and the rcmd. This
585 1       * below arg is sent to auxproc to tell auxproc which
586 1       * argument to update the format sting with the outputfd.
587 1       */

589 1      i = 3;                    /* For argv[3] -- See outputfd_fmt below.*/

591 1      memcpy(p, &i, sizeof(int));

593 1      p += sizeof (int);

595 1      /*
596 1       * the flags, which are always zero currently
597 1       */

599 1      i = 0;
600 1      memcpy(p, &i, sizeof (int));
601 1      p += sizeof (int);

603 1      /*
604 1       * The "xcpiogen command" that we will run locally.
605 1       */

607 1      (void)strcpy(p, rcx -> rc_cpiogen_executable);
608 1      p += strlen(p)+1;

610 1      /*
611 1       * The argc for the "xcpiogen command", and its argv vector.
612 1       */

614 1      memcpy(p, &xcpiogen_argc, sizeof (int));
615 1      p += sizeof (int);

617 1      /*
618 1       * The argv vector, which is argv0, the outputfd
619 1       * thingy, and the progress report mode flag.
620 1       */

622 1      (void)strcpy(p, xcpiogen_argv0);
623 1      p += strlen(p)+1;

625 1      (void)strcpy(p, submit_file_flag); /* xcpiogen argv[1] */
626 1      p += strlen(submit_file_flag) + 1;

628 1      (void)strcpy(p, temp_submit_file); /* xcpiogen argv[2] */
629 1      p += strlen(temp_submit_file) + 1;

631 1      (void)strcpy(p, outputfd_fmt); /* xcpiogen argv[3] */
632 1      p += strlen(outputfd_fmt) + 1;
```

```
634 1      (void)strcpy(p, progress_report_flag); /* xcpiogen argv[4] */
635 1      p += strlen(progress_report_flag) + 1;

637 1      if (bufsizeP != NULL)
638 2      {
639 2           (void)strcpy(p,bufsizeP); /* xcpiogen argv[??] */
640 2           p += strlen(bufsizeP) + 1;
641 1      }

643 1      /*
644 1       * Follow this with the total bytes flag and value.
645 1       */

647 1      strcpy(p, total_bytes_flag); /* xcpiogen argv[??] */
648 1      p += strlen(total_bytes_flag) + 1;
649 1      strcpy(p, total_bytes_stringP); /* xcpiogen argv[??] */
650 1      p += strlen(total_bytes_stringP) + 1;

652 1      /*
653 1       * socket info for db API
654 1       */

656 1      (void)memcpy(p, (char *)&temp_socket_port, sizeof (int));
657 1      p += sizeof (int);

659 1      /*
660 1       * socket host name for db API
661 1       */

663 1      (void)strcpy(p, temp_socket_host);
664 1      p += strlen(p)+1;

666 1      if (NULL != pi) /* send filespec */
667 2      {
668 2           (void)strcpy(p, pi->list);
669 2           p += strlen(p)+1;
670 1      }
671 1      else
672 2      {
673 2           *p++ = 0;
674 1      }

676 1      /*
677 1       * workitem name
678 1       */

680 1      (void)strcpy(p, temp_workitem_name);
681 1      p += strlen(p)+1;

683 1      /*
684 1       * assert that our arithmetic above was done correctly
685 1       */

687 1      if ((size_t)(p - auxproc_databuf) != data_len)
688 2      {
689 2           rbe_log_stats(0, "assertion failed: cmd size miscount");
690 2           return -1;
691 1      }

693 1      /*
694 1       * Send the restore command to auxproc. Auxproc will start
695 1       * The remote command (if necessary) and xcpiogen.
696 1       */

698 1      auxcmdpacket(xp-> xp_fd_to_x,
```



```
699 1                'r', (int)data_len, auxproc_databuf);

701 1      /*
702 1       * Obtain the fork status
703 1       */

705 1      resfd = xp-> xp_fd_from_x;

707 1      i = auxresults(resfd, '0', 0, &resultsbufptr);

709 1      if (i < 0)
710 2      {
711 2           rbe_log_stats (0,
712 2                          "** Error while starting auxproc for work item
                                                                    \"%s\"",
713 2                          temp_workitem_name);
714 2           null_free (resultsbufptr);
715 2           return -1;
716 1      }

718 1      /* This memory management is crap */

720 1      pkt0p = &pkt0p_buffer;
721 1      memcpy(pkt0p, resultsbufptr, sizeof(pkt0p_buffer));
722 1      memcpy(results, resultsbufptr, sizeof(pkt0p_buffer));

724 1      if ((pkt0p->msglen) > 0)
725 2      {
726 2           errstr = resultsbufptr + sizeof *pkt0p;
727 2           *err_str = esl_strdup(errstr);
728 1      }
729 1      else
730 2      {
731 2           errstr = "";
732 2           *err_str = esl_strdup("");
733 1      }

735 1      /*
736 1       * if the fork failed, the cpiogen start fails
737 1       */

739 1      if (0 != pkt0p->failcode)
740 2      {
741 2           int jnk;


744 2           if (strlen(errstr) > (size_t)0)
745 3           {
746 3               rbe_log_stats(0,
747 3                             "** Error while starting auxproc, error %s,
                                                                    "
748 3                        "for work item \"%s\"" ,errstr,
                                                    temp_workitem_name);
749 2           }

751 2           free (resultsbufptr);

753 2           /*
754 2            * collect the useless 'r' reply packet
755 2            */

757 2           resultsbufptr = (char *)&jnk;
758 2           (void)auxresults(resfd, 'r', sizeof (int), &resultsbufptr);
759 2           return -1;
760 1      }
```

```
762 1     /*
763 1      * Caller assumes responsibility for (eventually)
764 1      * collected exit status of remote and local programs.
765 1      */

767 1     pid = pkt0p->pid;

769 1     free (resultsbufptr);

771 1     return pid;
772   }         /* end of start_cpiogen() */
```



```
775   /*
776    * 1) Remove rcx references.
777    */

779   static char *
780   make_remote_cpiogen_cmd(struct recover_context *rcx,
781                           int SubmitObjectID,
782                           int SubmitElemID)
783 1 {
784 1     char *rcmdpath = generate_rcmdpath(SubmitObjectID,
785 1                                        SubmitElemID);
786 1     char *minus_c = "";
787 1     char *minus_c_arg = "";
788 1     char *noclobber = "";
789 1     char *remdebugflag = "";
790 1     char *cmd;
791 1     unsigned len;
792 1     RBC_WORKITEM *work_itemP;
793 1     char *bufsizeP = "";
794 1     char bufsize_buffer[20];

796 1     char temp_dirtop[SUBMIT_FIELD_MAX];
797 1     char temp_workitem_name[SUBMIT_FIELD_MAX];
798 1     OverwritePolicy temp_overwrite_policy;
799 1     int GetSEStatus = 0;

801 1     if (NULL == rcmdpath)
802 2     {
803 2       return NULL;
804 1     }
805 1     temp_overwrite_policy = GetSEDestOverWritePolicy(SubmitObjectID,
806 1                                                      SubmitElemID,
807 1                                                      &GetSEStatus);

809 1     if (0 != GetSEStatus)
810 2     {
811 2       rbe_log_stats(0, "Unable to get overwrite policy.");
812 2       return NULL;
813 1     }

815 1     if (GetSEDestDirTop(SubmitObjectID, SubmitElemID,
816 1                         temp_dirtop, SUBMIT_FIELD_MAX,
817 1                         &GetSEStatus) != 0)
818 2     {
819 2       rbe_log_stats(0, "Unable to get dirtop.");
820 2       return NULL;
821 1     }

823 1     if (GetSEWorkItemName(SubmitObjectID, SubmitElemID,
824 1                           temp_workitem_name,
825 1                           SUBMIT_FIELD_MAX,
826 1                           &GetSEStatus) != 0)
827 2     {
828 2       rbe_log_stats(0, "Unable to get work item name.");
829 2       return NULL;
830 1     }

832 1     if (temp_dirtop != NULL)
833 2     {
834 2          minus_c = " -c ";
835 2          minus_c_arg = temp_dirtop;
836 1     }

838 1     work_itemP = rbc_find_workitem_in_config(temp_workitem_name,
```

```
839 1                                                   NULL, NULL,
840 1                                                   rcx->rc_config, /* OK */
841 1
                                                    RBC_FIND_WORKITEM_NO_OPTIONS);
842 1       if (work_itemP != NULL
843 1            && DEFAULT_EB_BUFSIZE != work_itemP->recover_client_bufsize)
844 2       {
845 2            sprintf(bufsize_buffer, " -A -b -A %d",
846 2                    work_itemP->recover_client_bufsize);
847 2            bufsizeP = bufsize_buffer;
848 1       }

850 1       switch (temp_overwrite_policy)
851 2       {
852 2       case RC_OVERPOL_NO_CLOBBER:
853 2            noclobber = " -A -onever";
854 2            break;

856 2       case RC_OVERPOL_NEW_CLOBBER:
857 2            noclobber = " -A -onewer";
858 2            break;

860 2       default:
861 2            break;  /* do something better here */
862 1       }

864 1       if (debugmode)
865 2       {
866 2            static char debugarg[100];


869 2            (void)sprintf(debugarg, " -x    /tmp/RBRdebug%d     ", getpid(
                                                                          ));
870 2            remdebugflag = debugarg;
871 1       }

873 1       len = strlen(rcmdpath) +
874 1              strlen(bufsizeP) +
875 1              strlen(remdebugflag) +
876 1              strlen(minus_c) +
877 1              strlen(minus_c_arg) +
878 1              strlen(noclobber) +
879 1              1;     /* for '\0' */

881 1       cmd = sm_fmalloc(len);
882 1       (void)sprintf(cmd, "%s%s%s%s%s%s",
883 1                     rcmdpath,
884 1                     bufsizeP,
885 1                     remdebugflag,
886 1                     minus_c,
887 1                     minus_c_arg,
888 1                     noclobber);

890 1       return cmd;
891   }         /* end of make_remote_cpiogen_cmd() */
```



```
894       /*
895        * 1) Remove rcx references.
896        * 2) Research whether ebc_normalize updates.
897        */
898       /*
899        * Construct the path name of the remote command that
900        * will be executed on the destination client.
901        *
902        * Return ptr to constructed string.
903        * NOTE: caller must copy if string is to be preserved.
904        */

906       static char *
907       generate_rcmdpath(int SubmitObjectID,
908                         int SubmitElemID)
909   1   {
910   1       static char *mybuf = NULL;
911   1       size_t len_needed;
912   1       char *pph;    /* pointer to %h */
913   1       char *p;
914   1       char *q;
915   1       char *norm_host;

917   1       char temp_scriptname[SUBMIT_FIELD_MAX];
918   1       char temp_client_hostname[SUBMIT_FIELD_MAX];

920   1       int GetSEStatus = 0;

922   1       if ( GetSERcmdScriptName(SubmitObjectID, SubmitElemID,
923   1                                temp_scriptname, SUBMIT_FIELD_MAX,
924   1                                &GetSEStatus) != 0)
925   2       {
926   2         rbe_log_stats(0, "Unable to get rcmd script name.");
927   2         return NULL;
928   1       }

930   1       if ( GetSEDestClientName(SubmitObjectID, SubmitElemID,
931   1                                temp_client_hostname, SUBMIT_FIELD_MAX,
932   1                                &GetSEStatus) != 0)
933   2       {
934   2         rbe_log_stats(0, "Unable to get client destination name.");
935   2         return NULL;
936   1       }
937   1       /*
938   1        * If there is a %h in the rc_client_scriptname,
939   1        * that means put the hostname in there.
940   1        *
941   1        * Sorry, no escapes implemented.  You can't have an
942   1        * rc_client_scriptname with a literal %h in it.  Tough.
943   1        */

945   1       for (pph = temp_scriptname; *pph != '\0'; pph++)
946   2       {
947   2            if (*pph == '%' && *(pph+1) == 'h')
948   3            {
949   3                break;
950   2            }
951   1       }

953   1       /*
954   1        * no %h, just use the bare client scriptname
955   1        */

957   1       if (*pph == '\0')
```

```
958  2      {
959  2           return temp_scriptname;
960  1      }

962  1      /*
963  1       * there is a %h ... insert the destination client
964  1       * name into the rc_client_scriptname.  First
965  1       * compute how much storage will be needed to
966  1       * hold the result.
967  1       */

969  1      if (!ebc_can_it_be_normalized(temp_client_hostname))
970  2      {
971  2          rec_api_log_csm(SUB_CSM_NOMEM, NULL);
972  2           rbe_log_stats(
                          0, "Could not allocate memory generate_rcmdpath");
973  2           /*      StartDoneCallBack(EP_RB_RECOVER_FATALERR);*/
974  2           return NULL;
975  1      }

977  1      norm_host = ebc_normalize(temp_client_hostname);
978  1      len_needed =
979  1           strlen(temp_scriptname) - 2 +            /* -2: %h */
980  1           strlen(norm_host) + 1;
                                                         /* +1: '\0' */

982  1      if (mybuf == NULL || strlen(mybuf) < (size_t)(len_needed-1))
983  2      {
984  2           if (mybuf != NULL)
985  3           {
986  3               free(mybuf);
987  2           }

989  2           if ((mybuf = malloc((int)len_needed)) == NULL)
990  3           {
991  3              rec_api_log_csm(SUB_CSM_NOMEM, NULL);
992  3               rbe_log_stats(
                            0, "Could not allocate memory generate_rcmdpath");
993  3              return NULL;
994  2           }
995  1      }

997  1      q = mybuf;
998  1      for (q = mybuf, p = temp_scriptname; p < pph; q++, p++)
999  2      {
1000 2           *q = *p;
1001 1      }

1003 1      (void)strcpy(q, norm_host);
1004 1      (void)strcat(mybuf, pph+2);

1006 1      return mybuf;
1007   }         /* end of generate_rcmdpath() */
```



```
1009     int
1010     StartWorkItemRestore(struct recover_context *rcx,
1011                          struct auxproc *xp,
1012                          int SubmitObjectID,
1013                          int SubmitElemID)
1014 1   {
1015 1       char    *rcmdv[4];   /* 0 hostname, 1 locuser, 2 remuser, 3 cmd */
1016 1       int      changed_priv = 0;
1017 1       int      xcpiogen_pid;
1018 1       rcmd_pkt0_info pkt0p_buffer;
1019 1       char *err_str_buffer = NULL;

1021 1       char temp_client_hostname[SUBMIT_FIELD_MAX];
1022 1       char temp_client_rbuname[SUBMIT_FIELD_MAX];
1023 1       long  temp_effective_uid;
1024 1       boolean_ty temp_src_sysadmin;

1026 1       int GetSEStatus = 0;
1027 1       int GetSOStatus = 0;


1030 1       if ( GetSEDestClientName(SubmitObjectID, SubmitElemID,
1031 1                                temp_client_hostname,
1032 1                                SUBMIT_FIELD_MAX,
1033 1                                &GetSEStatus) != 0)
1034 2       {
1035 2         rbe_log_stats(0, "Unable to get client destination name.");
1036 2         return -1;
1037 1       }



1041 1       if ( GetSEClientUserName(SubmitObjectID, SubmitElemID,
1042 1                                temp_client_rbuname,
1043 1                                SUBMIT_FIELD_MAX,
1044 1                                &GetSEStatus) != 0)
1045 2       {
1046 2         rbe_log_stats(0, "Unable to get client user name.");
1047 2         return -1;
1048 1       }



1052 1       temp_src_sysadmin = GetSOSourceSystemAdmin(SubmitObjectID,
1053 1                                                  &GetSOStatus);
1054 1       if (0 != GetSOStatus)
1055 2       {
1056 2         rbe_log_stats(
                    0, "Unable to get the source system admin priveledges.");
1057 2         return -1;
1058 1       }

1060 1       if ( GetSOEffectiveUID(SubmitObjectID,
1061 1                              &temp_effective_uid,
1062 1                              &GetSOStatus) != 0)
1063 2       {
1064 2         rbe_log_stats(0, "Unable to get effective uid.");
1065 2         return -1;
1066 1       }


1069 1       if (0 != (temp_src_sysadmin) || 0 != temp_effective_uid)
1070 2       {
1071 2            set_recovery_privileges(rcx, &changed_priv);
1072 1       }
```

```
1074  1      rcmdv[0] = temp_client_hostname;
1075  1      rcmdv[1] = temp_client_rbuname;
1076  1      rcmdv[2] = temp_client_rbuname;
1077  1      rcmdv[3] = make_remote_cpiogen_cmd(rcx,
1078  1                                        SubmitObjectID,
1079  1                                        SubmitElemID);

1081  1      if(NULL == rcmdv[3])
1082  2      {
1083  2        return -1;
1084  1      }
1085  1      xcpiogen_pid = start_cpiogen(rcx, xp,
1086  1                                   SubmitObjectID,
1087  1                                   SubmitElemID,
1088  1                                   0,
1089  1                                   rcmdv,
1090  1                                   &pkt0p_buffer,
1091  1                                   &err_str_buffer);

1093  1      if (changed_priv)
1094  2      {
1095  2           reset_recovery_privileges(rcx, changed_priv);
1096  1      }
1097  1      return xcpiogen_pid;
1098    } /* StartWorkItemRestore() */
```



```
1100    /*
1101     * Check to see if the user has root access to the
1102     * destination client. If so, give him/her the root
1103     * privileges on the recovery.
1104     */

1106    static void
1107    set_recovery_privileges(struct recover_context *rcx,
1108                            int *changed)
1109  1 {
1110  1      int root_access;
1111  1      eperrno errnum;


1114  1      *changed = 0;
1115  1      (void)rbc_canirecover(rcx->rc_config, rcx->rc_client_hostname,
1116  1                            rcx->rc_human_uidname, &root_access,
                                                                  &errnum);
1117  1      if (root_access)
1118  2      {
1119  2           if (debugmode)
1120  3           {
1121  3                rbe_log_stats(
                            0, "the user is a sys admin for the dest client");
1122  2           }

1124  2           rcx->rc_recovery_flags |= RC_RECFLAG_DEST_SYSADMIN;
1125  2           if (rcx->rc_effective_uid != 0)
1126  3           {
1127  3                if (debugmode)
1128  4                {
1129  4                     rbe_log_stats(0, "changing the uid to admin status");
1130  3                }

1132  3                rcx->rc_effective_uid = 0;
1133  3                rcx->rc_effective_uidname = "root";
1134  3                *changed = SET_ROOT;
1135  2           }
1136  1      }
1137  1      else
1138  2      {
1139  2           if (debugmode)
1140  3           {
1141  3                rbe_log_stats(
                       0, "the user don't have sys admin status on dest client");
1142  3                rbe_log_stats(
                                 0, "effect uid = %d", rcx->rc_effective_uid);
1143  2           }

1145  2           /*
1146  2            * The user does not have root access to the dest
1147  2            * client. But s/he is the system admin for the
1148  2            * source client, therefore, we need to stripe the
1149  2            * root stuff from the user during the recovery.
1150  2            */

1152  2           rcx->rc_recovery_flags ^= RC_RECFLAG_DEST_SYSADMIN;
1153  2           if (rcx->rc_recovery_flags & RC_RECFLAG_SOURCE_SYSADMIN)
1154  3           {
1155  3                if (debugmode)
1156  4                {
1157  4                     rbe_log_stats(
                                 0, "changing the uid to regular user status");
1158  3                }
```

```
1160  3                 rcx->rc_effective_uid = rcx->rc_human_uid;
1161  3                 rcx->rc_effective_uidname = rcx->rc_human_uidname;
1162  3                 *changed = SET_USER;
1163  2             }
1164  1         }
1165    }         /* end of set_recovery_privileges() */
```



```
1168    /*
1169     * Reset the user's identity. The user may have root access
1170     * on the destination client, but not on the source, and vice
1171     * versa.
1172     */

1174    static void
1175    reset_recovery_privileges(struct recover_context *rcx,
1176                              int changed)
1177  1 {
1178  1     if (changed == SET_ROOT)
1179  2     {
1180  2           if (debugmode)
1181  3           {
1182  3               rbe_log_stats(
                                0, "resetting the uid to regular user status");
1183  2           }

1185  2           rcx->rc_effective_uid = rcx->rc_human_uid;
1186  2           rcx->rc_effective_uidname = rcx->rc_human_uidname;
1187  1     }
1188  1     else
1189  2     {
1190  2           if (debugmode)
1191  3           {
1192  3               rbe_log_stats(0, "resetting the uid to sys admin status");
1193  2           }

1195  2           rcx->rc_effective_uid = 0;
1196  2           rcx->rc_effective_uidname = "root";
1197  1     }
1198    }         /* end of reset_recovery_privileges() */
```

```
1202  /**
1203   **
1204   **  FUNCTION DESCRIPTION:
1205   **
1206   **      This function will start the workitem restore termination.
1207   **
1208   **
1209   **  INPUTS:
1210   **      int auxproc_pid -- auxproc's pid.
1211   **
1212   **  RETURN VALUE:
1213   **
1214   **      none
1215   **
1216   **
1217   **  SIDE EFFECTS:
1218   **
1219   **       auxproc sent the USR1 signal, auxproc will send xcpiogen the
1220   **      TERM signal to quit the restore. Auxproc will wait until the
1221   **      restore terminates by waiting for the remote command's exit
1222   **      status.
1223   **
1224   **++
1225   **/

1227  void
1228  QuitWorkItemRestore(int auxproc_pid)
1229  1 {


1232  1     /* Auxproc will now alert xcpiogen by sending the
1233  1      * TERM signal.
1234  1      * This will give xcpiogen the ability to
1235  1      * clean up tmpfiles and clean-up sockets.
1236  1      * xcpiogen will commit suicide as a result of
1237  1      * this signal.
1238  1      */


1241  1     /*
1242  1      * Alert the auxproc that we want out.
1243  1      * recxcpio etc should also die as a result
1244  1      * of xx_read_or_die_xx() in recx.  The SIGUSR1
1245  1      * should not kill the auxproc itself, but only
1246  1      * notify auxproc of the diminishing restore.
1247  1      */

1249  1     (void)kill(auxproc_pid, SIGUSR1);

1251  1     if (debugmode)
1252  2     {
1253  2          rbe_log_stats(0, "%s %d quiting restore in process",
1254  2                        AUXPROCNAME,
1255  2                        auxproc_pid);
1256  1     }

1258  1     /*
1259  1      * now that we have indirectly killed the xcpiogen and alerted the
                                                                          a
1260  1      * auxproc,
                    the signal from the auxproc will be picked up by the next
1261  1      * routine, auxprocsig_handler, it will notify the user of the
1262  1      * results and does the cleaning up.
```



```
1263  1      */
1264  1     return;
1265  }         /* QuitWorkItemRestore() */
```

```
1267  /**  GetAuxprocResults()
1268   **
1269   **  FUNCTION DESCRIPTION:
1270   **
1271   **     Inheritted from auxprocsig_handler().
1272   **
1273   **      This routine is called when there is information appears
1274   **     in the aux-process. The aux-process starts xcpiogen and
1275   **      is responsible for "listening" to status coming from xcpiogen.
1276   **      When status comes back from xcpiogen, the aux-process signals
1277   **      this process, which is trapped by the caller. And finally,
1278   **      this routine is called.
1279   **
1280   **  Args:
1281   **    (I) resfd int -- results file decriptor from auxproc.
1282   **    (O) results -- work item results.
1283   **
1284   **  RETURN VALUE:
1285   **     The number of local and remote status collected from fd.
1286   **
1287   **  SIDE EFFECTS:
1288   **
1289   **      none
1290   **
1291   **
1292   **/


1295  int
1296  GetAuxprocResults(int resfd,
1297                    wi_restore_results *results)
1298 1 {
1299 1     int           exit_stat;
1300 1     char          *resbuf;
1301 1     int           n_read;
1302 1     char          c = 0;
1303 1     int          n_status_read = 0;

1305 1     while ((n_status_read < 2) && (1 == fd_avail_test(resfd)))
1306 2     {
1307 2       resbuf = (char *) &exit_stat;

1309 2       if(1 != pread_or_warn(resfd, &c, 1, auxproc_comm_warning))
1310 3       {
1311 3          return 0;
1312 2       }

1314 2       if (debugmode)
1315 3       {
1316 3          rbe_log_stats(0, "GetAuxprocResults() called: '%c'", c);
1317 2       }

1319 2       if (c == 'R')
1320 3       {
1321 3          if (1 == fd_avail_test(resfd))
1322 4          {
1323 4            /*
1324 4             * the 'R' command is the for the remote process status
1325 4             */
1326 4            n_read = auxres2(resfd, 'R', sizeof(int), &resbuf);
1327 4            if (-1 != n_read )
1328 5            {
1329 5              results -> remote_exit_status = exit_stat;
1330 5              results -> remote_exit_set = TRUE;
1331 5              n_status_read++;
```



```
1332 5              if (debugmode)
1333 6              {
1334 6                rbe_log_stats(
                            0, "remote exit status obtained: %d", exit_stat);
1335 5              }
1336 4            }
1337 4            else
1338 5            {
1339 5              rbe_log_stats(
                        0, "Internal error: remote exit status Incomplete.");
1340 5              return -1;
1341 4            }
1342 3          }
1343 2        }
1344 2        else if (c == 'r')
1345 3        {
1346 3           if (1 == fd_avail_test(resfd))
1347 4           {
1348 4             /*
1349 4              * the 'r' command is the for the local process status
1350 4              */

1352 4             n_read = auxres2(resfd, 'r', sizeof(int), &resbuf);
1353 4             if (-1 != n_read )
1354 5             {
1355 5               results -> local_exit_status = exit_stat;
1356 5               results -> local_exit_set = TRUE;
1357 5               n_status_read++;
1358 5               if (debugmode)
1359 6               {
1360 6                 rbe_log_stats(
                            0, "local exit status obtained: %d", exit_stat);
1361 5               }
1362 4             }
1363 4             else
1364 5             {
1365 5               rbe_log_stats(
                          0, "Internal error: local exit status Incomplete.");
1366 5               return -1;
1367 4             }
1368 3           }
1369 2        }
1370 2        /*      sleep (1);*/
1371 1     } /* while() */

1373 1     return n_status_read;
1374 1     /*return n_status_read;*/

1376  }         /* GetAuxprocResults() */
```

```
1379  /*
1380   * KillWorkItemRestore()
1381   *
1382   *  Kill the work item restore. Keep in mind this
1383   *  may only be done if the work item is not running
1384   *  a restore.
1385   *
1386   *  This routine also does the waitpid for auxproc.
1387   *  The waitpid (ChildDone()) eliminates the defunct auxproc
1388   *  process.
1389   *
1390   *  If the work item is running then one must do the
1391   *  following.
1392   *
1393   *  1) Call QuitWorkItemRestore()
1394   *  2) Wait for and read results from the cmd_from pipe.
1395   *  3) Call KillWorkItemRestore()
1396   *
1397   * Args:
1398   *  ap_pid -- auxproc pid.
1399   *  cmd_to -- auxproc cmd pipe.
1400   *
1401   * Returns:
1402   *  int -- Zero for success.
1403   */
1404  int
1405  KillWorkItemRestore(int ap_pid, int cmd_to)
1406 1 {
1407 1   int killRet;
1408 1   int apResult;
1409 1   char *apName=AUXPROCNAME;
1410 1   char *databuf = NULL;
1411 1   int ChildDoneRet;

1413 1   killRet = kill(ap_pid, SIGTERM);

1415 1   if (-1 == killRet)
1416 2   {
1417 2         rbe_log_stats(
                  0, "Can't send sigterm to \"%s\": Pid: %d, error = %s\n",
1418 2                       apName, ap_pid, strerror(errno));
1419 2         return -1;
1420 1   }

1422 1   do
1423 2   {
1424 2      ChildDoneRet = ChildDone(ap_pid, &apResult);
1425 2      if(0 == ChildDoneRet)  sleep (1);
1426 1   }
1427 1   while(0 == ChildDoneRet);

1429 1   switch (ChildDoneRet)
1430 2   {
1431 2     /*  -1 internal error. errno is set.
1432 2      *  0  child still running.
1433 2      *  1  child exited.
1434 2      *  2  child signalled (no core)
1435 2      *  3  child signalled core file generated.
1436 2      *  4  child stopped.
1437 2      */
1438 2       case(-1):

1440 2         return -1;
1441 2         /* no break necessary */
```



```
1443 2       case(1):

1445 2           rbe_log_stats(0,
1446 2                         "Sigterm did not bring down \"%s\": Pid: %d,"
1447 2                         "  it instead exitted with = %d\n",
1448 2                         apName, ap_pid, apResult);
1449 2           break;

1451 2       case(2):
1452 2       case(3):
1453 2         if (SIGTERM != apResult)
1454 3          {
1455 3           rbe_log_stats(0,
1456 3                         "Sigterm did not bring down \"%s\": Pid: %d,,"
1457 3                         "  instead killed by signal = %s\n",
1458 3                         apName, ap_pid,
1459 3                         strsignal(apResult));
1460 2         }
1461 2         break;

1463 2       case(4):
1464 2           rbe_log_stats(0,
1465 2                         "Sigterm did not bring down \"%s\": Pid: %d,"
1466 2                         "  instead stopped by signal = %s\n",
1467 2                         apName, ap_pid,
1468 2                         strsignal(apResult));

1470 2         break;

1472 1   }


1475 1   /*  auxcmdpacket(cmd_to, 'q', 0, databuf);*/
1476 1   return 0;

1478  } /* KillWorkItemRestore() */
```

```
/*
 *                  RSLauxmain.c
 *
 * Copyright (c) 1998, 1999 EMC Corporation.  All rights reserved.
 *
 *
 * Table of Contents:
 *--------------------
** static char *rb_getmethod(host, int *concurrency, int *client_type)
** int looprw(int fd, char *buf, int nbytes, int (*func)());
** int fd_avail_1_wait_intr(int fd);
 */

#define  _POSIX_SOURCE 1
#define  E_GRANDFATHER 1
#define TIME_STRUCT
#include <eb/eb_port.h>

#include <netdb.h>
#include <pwd.h>
#include <ctype.h>
#include <sys/wait.h>
#include <util/esl_select.h>
#include <eerrno/esl_strerror.h>
#include <util/esl_limit.h>
#include <stdlib.h>

/* SFP header */
#include <cdl/cdl.h>
#include <cdl/cdl_server.h>

#include <ebconfig/rbconfig.h>
#include <epffdb/ffdb.h>
#include <epffdb/rbfinclient.h>
#include <ebutil/eb_normalize.h>
#include <ebutil/ebsock_if.h>
#include <ebutil/ebutil.h>
#include <eb/rb_log.h>
#include <restore/REprogmsg.h>

#include <RSLinterns.h>
#include <RSLspexits.h>
#include <RSLremfd.h>
#include "RSLauxSupp.h"

/* EDMLINK API */
#include <edmlink/edmlink_api.h>


/* lint ugliness */
#ifndef Printf
#define Printf   (void)printf
#endif
#define MAX_FD 1024
int debugmode = 0;
int is_symmpath = 0;
int sp_cdlexitdone = 0;

static int xcpiogen_pid = -1;
static int xcpiogen_prog_fd = -1;
int logging_channel = -1;

int xcpiogen_pipe[2] = {-1,-1};
static char wifiledir[] = "/usr/epoch/etc/";

static char *rb_getmethod(
```



```
        register char *host, uint_t *concurrency, uint_t *client_type);
static void sigusr1_handler(int sigvalue);
                                                    /* "attention" signal */
static void sigterm_handler(int sigvalue);  /* sigterm signal */
static int write_CDL_no_eintr(int fd, char *buf, int nbytes);
static int read_CDL_no_eintr(int fd, char *buf, int nbytes);

static int ForwardXcpiogenProgress(int xcpiogen_prog_fd,
                                   int restore_engine_prog_fd,
                                   boolean_ty *zero_byte_read);

static int DemuxAuxChildren(int progress_fd,
                            int remote_fd,
                            char *remote_progname,
                            int xcpiogen_fd,
                            int xcpiogen_pid,
                            int *remote_exit);

extern int fd_avail_test(int); /* Found in RSLauxSupp.c */

struct auxproc_context
{
    int       ap_my_auxnum;
    int       ap_r_fd;
    int       ap_w_fd;
    int      ap_r_bulk_fd;
    int      ap_w_prog_fd;
    char     ap_cmd;
    int      ap_datalen;
    char    *ap_data;
    int      ap_resultlen;
    char    *ap_resultdata;
    ushort_t ap_shelltcp_port;
    int      ap_have_shelltcp_port;
    int      ap_socketport;
    char    *ap_sockethost;
    char    *ap_workitem;
    struct rbc_configs *ap_config;
    char     ap_error_message[4096];
};
/*
 * Read remote stderr output from fd.
 * Parse it; act on it according to protocol.
 * Remote exit status is the last thing that comes
 * back via the remote stderr stream; return the remote
 * exit status via *exitp.
 */

enum input_states
{
    INSTATE_NG,
    INSTATE_SEARCH_PREFIX0,
    INSTATE_SEARCH_PREFIXN,
    INSTATE_GATHER_COOKIE,
    INSTATE_SEARCH_SUFFIXN,
    INSTATE_GATHER_STATUS,
    INSTATE_COPY_TO_STDOUT,
    INSTATE_NEWLINE
};
static boolean_ty parse_remote_stderr_info2(int prog_fd,
                                            int remote_fd,
                                            int *exitp,
                                            char *remhostname,
                                            boolean_ty first_call,
                                            enum input_states
                                                *state_ptr,
```

```
                                        enum input_states
                                                *next_state_ptr,
                                        boolean_ty
                                    *skipping_leading_whitespace,
                                        int *parsePos,
                                        int *msgPos);


static char *recover_size_prefix(struct auxproc_context *cxp);
static int  *ebr_direct_rcmd(char **ahost, unsigned short inport,
                             struct auxproc_context *cxp,
                                                  char *locuser,
                             char *remuser, char *cmd, int *fd2p);
```



```
/*
 * The auxilliary process(es) communicate with the main
 * process via a simple protocol run on a pair of pipes.
 *
 * The parent writes commands to the auxilliary process.
 * The format of a command is:
 *
 *       <cmd><data-len><data>
 *
 * where
 *       <cmd> is a one-byte command
 *       <data-len> is an "int",
                                  and indicates the number of <data> bytes
 *       <data> is command-specific data.
 *
 * <data-len> may be zero, but must always be present. Therefore, the
 * minimum command "packet" is five bytes long.
 *
 * Result packets are written back to the parent.
                                                        The packet format is
 * the same as the command format, though (obviously) the format
 * of the data in a results packet is usually different from
 * the format in a command packet.
 *
 * Communications are assumed to be error-free.   In other
 * words, pipes are assumed to work correctly.
 */

static int attn_ec;      /* count of SIGUSR1s (ATTN signals) */



/*
 * 'z' prefixes identify top-level routines
 *     called from the auxproc switch.
 *
 *     There is no significant to the 'z' letter -- I just picked
 *     a letter at random (hah!) to identify the top level funcs.
 */

static void z_rcmdfilter(struct auxproc_context *cxp);
static void z_exec_separate_auxproc(struct auxproc_context *cxp,
                                    char *pathname);
```

```
183     int main(int argc, char *argv[])
184   1 {
185   1     char *auxproc_intial_delay_str = NULL;
186   1     int auxproc_intial_delay_int = 0;

188   1     if (argc != 9)
189   2     {
190   2          exit(1);
191   1     }

193   1     auxproc_intial_delay_str = getenv("AUXPROC_INITIAL_DELAY");
194   1     if (NULL != auxproc_intial_delay_str)
195   2     {
196   2       auxproc_intial_delay_int = atoi(auxproc_intial_delay_str);
197   2       sleep(auxproc_intial_delay_int);
198   1     }

200   1     debugmode = atoi(argv[6]);

202   1     do_auxproc(atoi(argv[1]), /* auxproc ordinate. */
203   1                atoi(argv[2]), /* cmd_to auxproc pipe*/
204   1                atoi(argv[3]), /* cmd_from auxproc pipe*/
205   1                atoi(argv[4]), /* bulk_to auxproc pipe*/
206   1                atoi(argv[5]), /* prog_from auxproc pipe*/
207   1                argv[0],       /* progname */
208   1                argv[7],       /* socket host name */
209   1                atoi(argv[8]));/* socket port */

211   1     return(0); /* Can we get here ?? */

213     } /* End main() */
```



```
215     /*
216      * Worker-bee loop.
217      *
218      * Read instructions on r_fd.
219      * Write results to w_fd.
220      */

222     int ncmds_rcvd = 0;

224     void
225     do_auxproc(int procnum,
226                int r_fd,
227                int w_fd,
228                int r_bulk_fd,
229                int w_prog_fd,
230                char *Xname,
231                char *sockethost,
232                int socketport)
233   1 {
234   1     struct auxproc_context    ctx;
235   1     struct servent           *sp;
236   1     sigset_t                  empty_set;
237   1     int                       rotation_size;
238   1     int                       errnum;
239   1     int                       index_fd;

241   1     /*
242   1      * close all file descriptors that we do not need
243   1      * we only need the ones passed to us from the restore
244   1      * engine.  So we can close anything above stderr and anything
245   1      * that is no equal to what was passed in
246   1      */
247   1     for(index_fd = 3; MAX_FD > index_fd; index_fd++)
248   2       {

250   2           if ((index_fd == r_fd) ||
251   2               (index_fd == w_fd) ||
252   2               (index_fd == r_bulk_fd) ||
253   2               (index_fd == w_prog_fd))
254   3             {
255   3               continue;
256   2             }
257   2           (void)close(index_fd);

259   1       }

261   1     memset(&ctx, 0, sizeof(struct auxproc_context));
262   1     /*
263   1      * Prepare sigaction parameters
264   1      */

266   1     sigemptyset(&empty_set);

268   1     /*
269   1      * ignore mild keyboard interrupts (^C); parent handles
270   1      */

272   1     eb_set_signal_handler(SIGINT, SIG_IGN, &empty_set, E_SA_RESTART);

274   1     /*
275   1      * SIGUSR1 used to get our attention when the parent
276   1      * process really wants us to stop what we are doing.
277   1      */

279   1     eb_set_signal_handler(
```

```
                    SIGUSR1, sigusr1_handler, &empty_set, E_SA_RESTART);
    eb_set_signal_handler(
                    SIGTERM, sigterm_handler, &empty_set, E_SA_RESTART);


    /*
     * Arguments passed are collected together in
     * this context structure only because that makes
     * it easier to add/change arguments in the future
     * (diddle the structure rather than diddle a zillion
     * calls in the switch statement).
     */

    ctx.ap_my_auxnum   = procnum;
    ctx.ap_r_fd        = r_fd;
    ctx.ap_w_fd        = w_fd;
    ctx.ap_r_bulk_fd   = r_bulk_fd;
    ctx.ap_w_prog_fd   = w_prog_fd;
    ctx.ap_sockethost  = sockethost;
    ctx.ap_socketport  = socketport;


    /*
     * Will need the shell/tcp service later; if
     * can't find it now we might as well complain now.
     */

    ctx.ap_have_shelltcp_port = 0;
    if ((sp = getservbyname("shell", "tcp")) != NULL)
    {
         ctx.ap_shelltcp_port = (ushort_t)sp->s_port;
         ctx.ap_have_shelltcp_port = 1;
    }
    else
    {

         WriteFmtStringMsg(
                    ctx.ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                         "Error: Cannot find shell/tcp network
                                                          service.\n"
                         "Browsing of catalogs may continue, but"
                         "the \"start\" command will not work.");
    }

    if (NULL == ctx.ap_config)
    {
        if(NULL == (ctx.ap_config = malloc(sizeof(
                                             struct rbc_configs))))
         {
           /* We would prefer to log this stuff, but we
            *  have not opened logging yet.
            */
           WriteFmtStringMsg(
                    ctx.ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                           "Could not allocate memory in
                                                        do_auxproc");
           exit(1);
         }
         if (rbc_parse_config(
                    NULL /*use the default name */ , &ctx.ap_config,
                           RBC_PARSE_DO_NOT_PRESERVE |
                                             RBC_PARSE_APPLY) != 0)


         {
           /* We would prefer to log this stuff, but we
            *  have not opened logging yet.
            */
```



```
           rec_api_log_csm(SUB_CSM_NO_PARSE_CFG, NULL);
           WriteFmtStringMsg(
                    ctx.ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                           "Auxproc -- Cannot parse configuration
                                                              file");
           exit(1);
         }
    }

    if ((errnum = eb_path_init()) != 0)
    {
      exit(1);
    }

    (void)rbe_log_init("auxproc");

    /*
     * Determine the "recoveries.log" file rotation size.
     * Use a default unless this was specified in the config file.
     */

    rotation_size = RBRECOVER_LOGSIZE;
    if ((ctx.ap_config != (struct rbc_configs *)NULL)
         && (ctx.ap_config->srlfile.filename != NULL))
    {
         rotation_size = ctx.ap_config->srlfile.rotation_size;
    }

    (void)rbe_log_add(STATS_LOGGING, eb_recover_logpath, LOGT_FILE,
                      rotation_size, &logging_channel);



    /*
     * The actual worker-bee loop
     */

    for (;;)
    {
         char c;
         int datalen;
#define SMALL_DATALEN 128
         char buf[SMALL_DATALEN];
         char *data;

         /*
          * read command byte
          */

         pread_or_die(r_fd, &c, 1, _exit);

         /*
          * read data-len
          */

         pread_or_die(r_fd, (char *)&datalen, sizeof datalen, _exit);

         /*
          * We do the read here for simple commands that do not
          * take much data.
          */

         if (datalen > SMALL_DATALEN || datalen == 0)
         {
             data = NULL;
         }
         else
```

```
401  3          {
402  3              pread_or_die(r_fd, buf, datalen, _exit);
403  3              data = buf;
404  2          }

406  2          ++ncmds_rcvd;
407  2          ctx.ap_cmd         = c;
408  2          ctx.ap_datalen     = datalen;
409  2          ctx.ap_data        = data;
410  2          ctx.ap_resultlen   = 0;
411  2          ctx.ap_resultdata  = NULL;

413  2          /*
414  2           * NOTE: This switch statement is in alpha-order
415  2           *       (case-folded) to make it easier for humans
416  2           *       to find entries (at least, it was in alpha-order
417  2           *       when I wrote this comment).
418  2           */

420  2          switch (c)
421  3          {
422  3              /*
423  3               * 'P'
424  3               *  Ping command.  Used for debugging.
425  3               */

427  3          case 'P':
428  3              ctx.ap_resultlen  = ctx.ap_datalen;
429  3              ctx.ap_resultdata = ctx.ap_data;
430  3              break;

432  3              /* 'q'
433  3               *  Quit command.
434  3               */

436  3          case 'q':
437  3              rbe_close_logs(logging_channel);
438  3              _exit(0);

440  3              /*
441  3               * 'r'
442  3               *  rcmd filter.  Fire up a process, typically xcpiogen,
443  3               *  with output from the process going to an rcmd
444  3               *  connection.
445  3               */

447  3          case 'r':
448  3              z_rcmdfilter(&ctx);
449  3              break;

451  3              /* 'X'
452  3               *  Exec separate process version of do_auxproc.
453  3               *  Used for debugging.
454  3               *  No value returned to parent.
455  3               *  If exec fails, auxproc dies.
456  3               */

458  3          case 'X':
459  3              z_exec_separate_auxproc(&ctx, Xname);
460  3              _exit(2);
461  3              break;

463  3          default:
464  3              rec_api_log_csm(SUB_CSM_INV_AUXPROC_CMD, NULL);
465  3              rbe_log_stats(
                            0, "auxproc -- invalid command in do_auxproc");
```



```
466  3              exit(1);
467  3              break;
468  2          }   /* end of switch */

470  2          /*
471  2           * Simple commands set ap_resultlen and ap_resultdata
472  2           * and we push the results to the parent here.
473  2           */

475  2          if (ctx.ap_resultlen >= 0)
476  3          {
477  3              pwrite_or_die(w_fd, &c, 1, _exit);
478  3              pwrite_or_die(w_fd, (char *)&ctx.ap_resultlen,
479  3                            sizeof ctx.ap_resultlen, _exit);
480  3              pwrite_or_die(w_fd, ctx.ap_resultdata,
481  3                            ctx.ap_resultlen, _exit);
482  2          }
483  2  #if 0
484  2          if('r' == c)
485  3          {
486  3            rbe_log_stats(0, "Auxproc(%d) local exit is %d.", getpid(),
487  3                          ctx.ap_resultlen);

489  2          }
490  2  #endif
491  1      }   /* end of for loop */
492  }          /* end of do_auxproc() */
```

```
494  /*
495   * Fire up a process with stdin from parent and output to rcmd.
496   * Protocol traffic between "parent" (main process) and us:
497   *
498   *       'r' parent --> auxproc     contains local & remote cmd info
499   *       '0' auxproc --> parent     returns "setup" result, see below
500   *       'R' auxproc --> parent     returns remote exit status (1 int)
501   *       'r' auxproc --> parent     returns local exit status  (1 int)
502   *
503   * The '0' result packet describes the results of setting things up.
504   * It contains the following information:
505   *
506   *       success/failure : int    [ 0 is success,
                                         non-zero is a failure code ]
507   *       errnum          : int    [ 0 if no applicable errno code ]
508   *       pid             : int    [ process ID of local xcpiogen proc
                                                                      ]
509   *       msglen          : int    [ length, in bytes,
                                                        of following str ]
510   *       errstr          : string [ string describing failure ]
511   *
512   * If things were set up successfully, then two integer zero
513   * values, a non-zero pid, and one more zero (msglen) are
514   * sent in the '0' packet.
515   *
516   * The defined failure codes are:
517   *       TBD
518   *
519   * The 'R' result packet will not be sent if the '0' result
520   * indicates that an error occurred.  The '0' and 'r' result
521   * packets are always sent.
522   *
523   * Parent passes the following in the 'r' packet <data>:
524   *       (string) rcmd-hostname
525   *       (string) rcmd-locuser
526   *       (string) rcmd-remuser
527   *       (string) rcmd-cmd
528   *       (int)    setuid-val
529   *       (int)    filter-cmd-fd-info (explained below)
530   *       (int)    future-flags (flags for future hacks;
                                                    always zero now)
531   *       (string) filter-cmd
532   *       (int)    filter-cmd-argc
533   *       (string) filter-cmd-argv0
534   *       (string) ...
535   *       (string) filter-cmd-argvN
536   *      (int)    db API socket info
537   *      (string) db API socket host name
538   *      (string) filespec from workitem
539   *
540   * If filter-cmd-fd-info is -1, then stdout on the filter cmd
541   * is set up to go to the rcmd.
542   * Otherwise, a separate file descriptor (neither stdout nor stderr)
543   * is set up to go to the rcmd, and the argv element indicated
544   * by filter-cmd-fd-info is used as a sprintf template for passing
545   * the file descriptor number to the filter process.
546   *
547   * For example, if filter-cmd-fd-info is 1, then filter-cmd-argv1
548   * should be a sprintf format string, which will be given to
549   * sprintf along with one integer to pass the file descriptor number
550   * to the filter command.
551   *
552   * Items which are (string) are '\0' terminated.
553   *
554   * Parent is responsible for obeying the built-in size limits:
```



```
555   *     no more than 100 argv strings for filter-cmd
556   *     no more than 10000 total bytes of auxproc data
557   *     no more than 100 bytes of filter-cmd-fd-info argv
558   */


561  struct rcmd_pkt0_info
562  {
563      int failcode;
564      int errnum;
565      int pid;
566      int msglen;
567      /* variable length char string message follows */
568  };

570  /*
571   * Function to build command line prefix for adding environment
                                                                variable
572   * for log truncation
573   */

575  static char *
576  recover_size_prefix(struct auxproc_context *cxp)
577  {
578      static char lbuff[128];

580      if (cxp->ap_config->crlfile.rotation_size != NO_ROTATION)
581      {
582          sprintf(lbuff, "EB_MAX_CLIENT_LOG_SIZE=%d;
                                          export EB_MAX_CLIENT_LOG_SIZE;",
583                  cxp->ap_config->crlfile.rotation_size);
584      }
585      else
586      {
587          memset(lbuff, 0, sizeof(lbuff));
588      }

590      return lbuff;
591  }       /* recover_size_prefix */
```

```
593     static void
594     z_rcmdfilter(struct auxproc_context *cxp)
595   1 {
596   1     char        *data;
597   1     char        *rcmd_stuff[4];
                              /* 0 hostname, 1 locuser, 2 remuser, 3 cmd */
598   1     char        *human_name;
599   1     struct rcmd_pkt0_info pkt0;
600   1     char        *pkt0_errstr = "";
601   1     static int   r_resultdata;
602   1     int          filter_cmd_argc;
603   1     int          filter_cmd_fd_info;
604   1     int          filter_cmd_fdout;
605   1     char        *filter_cmd;
606   1     char        *method;
607   1     int           dbseqno=0;
608   1     int           socket_port;
609   1     char          socket_host[100];
610   1     char        *socket_file;
611   1     char          wifilename[200];
612   1     FILE        *infofd;
                                        /* items for socket info file --- */
613   1     static char readmode[] = "r";
614   1     char          cbuf[200];
615   1     char          listener_filename[EB_MAXPATHLEN];
616   1     char        *workitem;
617   1     int          rcmd_stderr = -1;
618   1     int         *rcmd_fd;
619   1     int          rcmd_fds[2];
620   1     int          i,n;
621   1     char         c;
622   1     char        *p2;
623   1     int          wait_result;
624   1     char         fdarg[100];              /* BUILT-IN LIMIT */
625   1     char        *filter_cmd_argv[100];    /* BUILT-IN LIMIT */
626   1     char         databuf[10000];          /* BUILT-IN LIMIT */
627   1     char        *filespec;
628   1     char          sslbuff[200];
                                      /* for command with ssl parameter */
629   1     char         holdbuff[1024];
                                           /* hold rbrclient ... command */

631   1     RBC_WORKITEM *pwi;
632   1     RBC_WORKGROUP *pwg;
633   1     boolean_ty xcpiogen_still_running;
634   1     e_errno_ty errnum;



638   1     /* EDMLINK handle */
639   1     ELinkHandlePtr_ty    pELinkHandle    = NULL;

641   1     /* EDMLINK objects */
642   1     ELinkTargetObjPtr_ty pELinkTargetObj = NULL;
643   1     ELinkUserIdObjPtr_ty pELinkUserIdObj = NULL;
644   1     ELinkCmdObjPtr_ty    pELinkCmdObj    = NULL;

646   1     /* EDMLINK global options */
647   1     unsigned long ELinkOptions = 0;

649   1     /* EDMLINK status */
650   1     int ELinkStatus = 0;
651   1     int ELinkStatus_errno = 0;

653   1     data = cxp->ap_data;
```



```
655   1     if (data == NULL)
656   2     {
657   2          /*
658   2           * command dispatcher did not read data for us,
659   2           * there was too much.  We must read it.
660   2           */

662   2          data = databuf;
663   2          pread_or_die(cxp->ap_r_fd, data, cxp->ap_datalen, _exit);
664   1     }

666   1     /*
667   1      * extract the 4 rcmd strings.
668   1      * [3] is the recxcpio cmd.
669   1      */

671   1     for (i = 0; i < 4; i++)
672   2     {
673   2          rcmd_stuff[i] = data;
674   2          data += strlen(rcmd_stuff[i])+1;
675   1     }

677   1     /*
678   1      * extract the remote uidname to be used
679   1      */

681   1     human_name = data;
682   1     data += strlen(human_name) + 1;

684   1     /*
685   1      * extract the filter-cmd-fd-info
686   1      */

688   1     memcpy(&filter_cmd_fd_info, data, sizeof (int));
689   1     data += sizeof (int);

691   1     /*
692   1      * skip the flags
693   1      */

695   1     data += sizeof (int);

697   1     /*
698   1      * extract the filter command
699   1      */

701   1     filter_cmd = data;
702   1     data += strlen(filter_cmd)+1;

704   1     memcpy(&filter_cmd_argc, data, sizeof filter_cmd_argc);
705   1     data += sizeof filter_cmd_argc;

707   1     for (i = 0; i < filter_cmd_argc; i++)
708   2     {
709   2          filter_cmd_argv[i] = data;
710   2          data += strlen(filter_cmd_argv[i])+1;
711   1     }

713   1     filter_cmd_argv[filter_cmd_argc] = NULL;

715   1     /*
716   1      * extract db API socket info
717   1      */

719   1     memcpy((char *)&socket_port, data, sizeof (int));
```

```
720 1     data += sizeof (int);

722 1     /*
723 1      * extract db API socket host name
724 1      */

726 1     socket_file = data;
727 1     data += strlen(socket_file) + 1;

729 1     /*
730 1      * extract filespec
731 1      */

733 1     filespec = data;
734 1     data += strlen(filespec) + 1;

736 1     /*
737 1      * extract workitem
738 1      */

740 1     workitem = data;
741 1     cxp->ap_workitem = workitem;
742 1     data += strlen(workitem) + 1;

744 1     pkt0.failcode = 0;
745 1     pkt0.errnum   = 0;
746 1     pkt0.pid      = -1;
747 1     pkt0.msglen   = -1;
748 1     pkt0_errstr   = "";

750 1     /*
751 1      * locate work item in config info
752 1      */

754 1     if (NULL == cxp->ap_config)
755 2     {
756 2         if(NULL == (cxp->ap_config = malloc(sizeof(
                                                 struct rbc_configs))))
757 3          {
758 3              rbe_log_stats(
                             0, "Could not allocate memory in z_rcmdfilter!");
759 3              exit(1);
760 2          }
761 2          if (rbc_parse_config(
                         NULL /*use the default name */ , &cxp->ap_config,
762 2                               RBC_PARSE_DO_NOT_PRESERVE |
                                                 RBC_PARSE_APPLY) != 0)

764 3          {
765 3              rec_api_log_csm(SUB_CSM_NO_PARSE_CFG, NULL);
766 3              rbe_log_stats(
                             0, "auxproc -- Cannot parse configuration file");
767 3              exit(1);
768 2          }
769 1     }

771 1     for (pwg = cxp->ap_config->pgrouplist; ; pwg = pwg->next)
772 2     {
773 2          if (pwg == (RBC_WORKGROUP *)NULL)
774 3          {
775 3              char csm_err_msg[256];

777 3              rbe_log_stats(
                    0, "\n ** No work item name \"%s\" in configuration file",
778 3                                cxp->ap_workitem);
779 3              sprintf(csm_err_msg,
```



```
780 3                         "Cannot restore. Work item not in eb.cfg (
                                                    No work item for %s.)",
781 3                         cxp->ap_workitem);
782 3                 rec_api_log_csm(SUB_CSM_NO_WITEM_CFG, csm_err_msg);
783 3                 rbe_log_stats(0, "Cannot continue without a work item");
784 3                 exit(1);
785 2             }

787 2             for (pwi = pwg->pwilist; NULL != pwi; pwi = pwi->next)
788 3              {
789 3                  if (0 == strcmp(pwi->name, cxp->ap_workitem))
790 4                  {
791 4                      goto gotit;
792 3                  }
793 2              }
794 1         }

796 1     gotit:
797 1         /*
798 1          * connect by direct or rsh methods
799 1          */

801 1         if ((method = rb_getmethod(rcmd_stuff[0], NULL, NULL)) == NULL)
802 2         {
803 2           WriteFmtStringMsg(
                        cxp ->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_WARNING, 0,
804 2                               "Unable to get connection method to host
                                                          \"%s\": "
805 2                               "defaulting to \"rsh\" method\n",
806 2                               rcmd_stuff[0]);
807 2           method = "rsh";
808 1         }

810 1         /*
811 1          * now check for a work item override of the connection method
812 1          */

814 1         if (CNCTN_RSH != pwi->connection_type) /* if other than rsh */
815 2         {
816 2             switch (pwi->connection_type)
817 3              {
818 3              case CNCTN_RSH:
819 3                  method = "rsh";
820 3                  break;

822 3              case CNCTN_EDMLINK:
823 3                  method = "edmlink";
824 3                  break;

826 3              case CNCTN_DIRECT:
827 3                  method = "direct";
828 3                  break;

830 3              case CNCTN_SOCKET:
831 3                  method = "socket";
832 3                  break;

834 3              case CNCTN_NETWARE:
835 3                  method = "netware";
836 3                  break;

838 3              default:
839 3                  method = "???";
840 3                  break;
841 2              }
```

```
843  2          if (debugmode)
844  3           {
845  3              rbe_log_stats(
  0, "using connection method \"%s\" due to work item \"%s\" override",
846  3                      method, pwi->name);
847  2           }
848  1      }

850  1      if (NULL != pwi && pwi->recovery_init) /* if a recovery init
                                                      command specified */
851  2      {
852  2           (void)system(pwi->recovery_init);
853  1      }

855  1      /* OSGsw34952 -- Workaround for STPclose() behavioral defect
856  1         If pwi->ssl_groupname is not NULL we are going to be using
857  1         SP for the restore.
858  1         Shut off the SSL (STP) exit handler.

860  1         Environment variable is used here so the call to
861  1         CDL_noatexit can be avoided, if desired.
862  1      */

864  1      if ( NULL == getenv("AUXPROC_DONOT_PERFORM_CDLNOATEXIT") )
865  2      {
866  2        if ( NULL != pwi->ssl_groupname )
867  3        {

869  3  #if 0
870  3           if ( NULL != getenv("EDM_STP_LOGGING"))
871  4           {
872  4             (void) putenv("STP_EVTLEVEL=DEBUG2");
873  4             (void) putenv("STP_EVTLOG_SHARE=OFF");
874  3           }
875  3  #endif

877  3           (void) CDL_noatexit();
878  3           is_symmpath = 1;
879  2        }
880  1      }

882  1      if (0 == strcmp(method, "direct"))
883  2      {
884  2           /*
885  2            * direct connection method here
886  2            */

888  2           if (debugmode)
889  3            {
890  3                rbe_log_stats(0,"Invoking cmd with %s, %s, %s, %s\n",
891  3                              rcmd_stuff[0], rcmd_stuff[1],
892  3                              rcmd_stuff[2], rcmd_stuff[3]);
893  2            }

895  2           strcpy(cbuf,  recover_size_prefix(cxp));

897  2           if (strlen(cbuf) > (size_t)0)
898  3            {
899  3                sprintf(holdbuff, "( %s %s )", cbuf, rcmd_stuff[3]);
900  3                rcmd_stuff[3] = holdbuff;
901  2            }

903  2           rcmd_fd = ebr_direct_rcmd(
                              &rcmd_stuff[0], cxp->ap_shelltcp_port, cxp,
904  2                                rcmd_stuff[1], rcmd_stuff[2],
                                                          rcmd_stuff[3],
```



```
905  2                                          &rcmd_stderr);
906  2           if (debugmode)
907  3            {
908  3                rbe_log_stats(
                            0, "AUXPROC: rcmd returned fd %d, stderr fd %d\n",
909  3                                 ((
                            NULL != rcmd_fd) ? rcmd_fd[0] : -1), rcmd_stderr);
910  2            }

912  2           if (rcmd_fd == NULL)
913  3            {
914  3                pkt0.failcode = 2;
915  3                pkt0_errstr = "cannot set up remote connection";

917  3                goto send_0;
918  2            }
919  1      }
920  1      else if ( (0 == strcmp(method, "rsh")) || (0 == strcmp(
                                                        method, "edmlink")) )
921  2      {
922  2           /* variable for dealing with symm path and cross restore */

924  2           char sslName[CDL_HOST_LENGTH];
925  2           char sslGroup[CDL_STG_LENGTH];
926  2           char **stgNames;     /* for returned array of STG names */
927  2           int  numGroups = 0;
                                       /* for returned number of STG names */
928  2           int  addIndex = 0;
929  2           int  rc = 0;
930  2           boolean_ty  symmPathOK = FALSE;
931  2           /*
932  2            * rsh connection method here
933  2            */

935  2          /*
936  2            * Set up the rcmd connection (rsh method) to the client.
937  2            */

939  2          if (! cxp->ap_have_shelltcp_port)
940  3           {
941  3               struct servent *sp;

943  3               /*
944  3                * Try to get the port number again.
945  3                */

947  3               if ((sp = getservbyname("shell", "tcp")) != NULL)
948  4               {
949  4                   cxp->ap_shelltcp_port = (ushort_t)sp->s_port;
950  4                   cxp->ap_have_shelltcp_port = 1;
951  3               }
952  3               else
953  4               {
954  4                   pkt0.failcode = 1;
955  4                   pkt0_errstr = "shell/tcp service not found";
956  4                   goto send_0;
957  3               }
958  2           }

960  2           /*
961  2            * check for SSL enabled work item and if present add
962  2            * SSL information to command string as an environment
963  2            * variable--but first check for cross restore and make
964  2            * appropriate adjustments
965  2            */
```

```
967  2          if (pwi->ssl_groupname != NULL)
968  3          {
969  3              if (!ebc_same_host(
                           rcmd_stuff[0], pwi->sysname)) /* if x-recovery */
970  4              {
971  4                    /* Grab the STG entries for this client */
972  4                    rc = CDL_getavailablegroups(
                                    rcmd_stuff[0], &numGroups, &stgNames);
973  4                    if ((numGroups==0) || (rc < 0))
974  5                    {
975  5                      char errStr[128] ;
976  5                      sprintf(
    errStr, "Unable to use SymmPath for cross restore to host %s--using
                                                  network", rcmd_stuff[0]);
977  5                      rbe_user_error(0, errStr);
978  5                      /*
979  5                       * Default to network by leaving SP Flag set to
                                                                  False.
980  5                       */
981  4                    }
982  4                    else
983  5                    {
984  5                        symmPathOK = TRUE;
985  5                       /*
986  5                       * The destination client is SP enabled.
                                                                First check
987  5                       * to see if the same STG group exists.
                                                                If we don't
988  5                       * find the same one we will use the first,
                                                                  so set
989  5                       * that as a default
990  5                       */

992  5                        memset(sslGroup,0,CDL_STG_LENGTH);
993  5                        strncpy(sslGroup,stgNames[0],CDL_STG_LENGTH-1);

995  5                       /* Initialize the index */
996  5                       addIndex = numGroups;
997  5                       while (--addIndex >= 0)
998  5                          if (0 == strcmp(
                                    pwi->ssl_groupname, stgNames[addIndex]))
999  6                          {
1000 6                           strncpy(
                          sslGroup,stgNames[addIndex],CDL_STG_LENGTH-1);
1001 6                           continue;
1002 5                        }


1005 5                     sprintf(sslbuff, "EB_SSL_RECOVER=\"%s\" ;
                                          export EB_SSL_RECOVER ;  ",
1006 5                          sslGroup);


1009 4                  }
1010 3              }
1011 3              else  /* not a cross recovery--just normal SP */
1012 4              {
1013 4                   symmPathOK = TRUE;
1014 4                   memset(sslGroup,0,CDL_STG_LENGTH);
1015 4                   strncpy(
                             sslGroup,pwi->ssl_groupname,CDL_STG_LENGTH-1);
1016 4                   sprintf(sslbuff, "EB_SSL_RECOVER=\"%s\" ;
                                          export EB_SSL_RECOVER ;  ",
1017 4                       sslGroup);

1019 3              }
```



```
1021 2          }




1025 2          if (debugmode)
1026 3          {
1027 3              sprintf(cxp->ap_error_message,
1028 3                      "Calling ELinkShell with %s, %d, %s, %s, %s",
1029 3                      rcmd_stuff[0], cxp->ap_shelltcp_port,
                                                          rcmd_stuff[1],
1030 3                      rcmd_stuff[2], rcmd_stuff[3]);

1032 3              rbe_log_stats(0, "%s", cxp->ap_error_message);
1033 3              cxp->ap_error_message[0] = 0;
1034 2          }

1036 2          strcpy(cbuf,  recover_size_prefix(cxp));

1038 2          if (strlen(cbuf) > (size_t)0)
1039 3          {
1040 3              if ( symmPathOK == TRUE)
1041 4              {
1042 4                  sprintf(holdbuff, "(
                                %s %s %s)", cbuf, sslbuff, rcmd_stuff[3]);
1043 4                  rcmd_stuff[3] = holdbuff;
1044 3              }
1045 3              else
1046 4              {
1047 4                  sprintf(holdbuff, "( %s %s )", cbuf, rcmd_stuff[3]);
1048 4                  rcmd_stuff[3] = holdbuff;
1049 3              }

1051 2          }
1052 2          else if (symmPathOK == TRUE)
1053 3          {
1054 3              sprintf(holdbuff, "( %s %s)", sslbuff, rcmd_stuff[3]);
1055 3              rcmd_stuff[3] = holdbuff;
1056 2          }

1058 2          /* if not Symm Path and cbuf is NULL rcmd_stuff[3] stays
                                                              unchanged */



1061 2          /*
1062 2          ** EDMLINK API
1063 2          */

1065 2          /* set the initial edmlink transport methods */
1066 2          ELinkOptions = ELINK_SHELL_RCMD | ELINK_SHELL_REXEC;

1068 2          /* enable the edmlink transport method */
1069 2          if( 0 == strcmp(method, "edmlink") )
1070 3          {
1071 3              ELinkOptions |= ELINK_SHELL_EDMLINK;
1072 2          }

1074 2          if( debugmode )
1075 3          {
1076 3              /* turn on edmlink debugging */
1077 3              ELinkOptions |= ELINK_LOGLVL_DEBUG;
1078 2          }

1080 2          /* initialize the edmlink api */
1081 2          pELinkHandle = ELinkInitAPI( ELinkOptions );
```

```
1083  2          if( NULL == pELinkHandle )
1084  3          {
1085  3              sprintf(cxp->ap_error_message,
1086  3                      "ERROR: Could not initialize the EDMLINK API.");

1088  3              WriteFmtStringMsg(cxp-> ap_w_prog_fd,
1089  3                                EDMREPROGMSG_AUXPROC_ERROR, 0,
1090  3                                "%s\n", cxp->ap_error_message);

1092  3              rbe_log_stats(0, "%s", cxp->ap_error_message);
1093  3              return;
1094  2          }

1096  2          /* get new target host object */
1097  2          pELinkTargetObj = ELinkNewTargetObj( pELinkHandle,
1098  2                  rcmd_stuff[0] );

1100  2          if( NULL == pELinkTargetObj )
1101  3          {
1102  3              sprintf(cxp->ap_error_message,
1103  3                      "ERROR: Could not create a new EDMLINK target
                                                                  object.");

1105  3              WriteFmtStringMsg(cxp-> ap_w_prog_fd,
1106  3                                EDMREPROGMSG_AUXPROC_ERROR, 0,
1107  3                                "%s\n", cxp->ap_error_message);

1109  3              rbe_log_stats(0, "%s", cxp->ap_error_message);
1110  3              /* clean up and return */
1111  3              (void) ELinkDoneAPI( pELinkHandle );
1112  3              return;
1113  2          }

1115  2          /* get new user id object */
1116  2          pELinkUserIdObj = ELinkNewEbrUserIdObj( pELinkHandle,
1117  2                                                  pELinkTargetObj,
1118  2                                                  rcmd_stuff[1] );

1120  2          if( NULL == pELinkUserIdObj )
1121  3          {
1122  3              WriteFmtStringMsg(cxp-> ap_w_prog_fd,
1123  3                                EDMREPROGMSG_AUXPROC_ERROR, 0,
1124  3                                "Could not create a new EDMLINK user id
                                                                  object.\n" );
1125  3              /* clean up and return */
1126  3              if( NULL != pELinkTargetObj )
1127  3                  (void) ELinkDestroyObj(
                                        pELinkHandle, pELinkTargetObj );
1128  3              (void) ELinkDoneAPI( pELinkHandle );
1129  3              return;
1130  2          }

1132  2          /* get new command object */
1133  2          pELinkCmdObj = ELinkNewCmdObj( pELinkHandle,
1134  2                  pELinkTargetObj,
1135  2                  rcmd_stuff[3] );

1137  2          if( NULL == pELinkCmdObj )
1138  3          {
1139  3              sprintf(cxp->ap_error_message,
1140  3                      "ERROR: Could not create a new EDMLINK command
                                                                  object.");

1142  3              WriteFmtStringMsg(cxp-> ap_w_prog_fd,
1143  3                                EDMREPROGMSG_AUXPROC_ERROR, 0,
1144  3                                "%s\n", cxp->ap_error_message);
```



```
1146  3              rbe_log_stats(0, "%s", cxp->ap_error_message);
1147  3              /* clean up and return */
1148  3              if( NULL != pELinkTargetObj )
1149  3                  (void) ELinkDestroyObj(
                                        pELinkHandle, pELinkTargetObj );
1150  3              if( NULL != pELinkUserIdObj )
1151  3                  (void) ELinkDestroyObj(
                                        pELinkHandle, pELinkUserIdObj );
1152  3              (void) ELinkDoneAPI( pELinkHandle );
1153  3              return;
1154  2          }

1156  2          rcmd_fd = rcmd_fds; /* set valid pointer */

1158  2          ELinkStatus = ELinkShell( pELinkHandle,
1159  2                  pELinkTargetObj,
1160  2                  pELinkUserIdObj,
1161  2                  pELinkCmdObj,
1162  2                  &rcmd_fd[0],
1163  2                  &rcmd_stderr );
1164  2          ELinkStatus_errno = errno;

1166  2          rcmd_fd[1] = rcmd_fd[0]; /* this one is bi-directional */

1168  2          if (debugmode)
1169  3          {
1170  3              rbe_log_stats(
                            0, "AUXPROC: ELinkShell returned %d, errno %d,"
1171  3                           " fd %d, stderr fd %d\n", ELinkStatus,
1172  3                           ELinkStatus_errno, rcmd_fd[0], rcmd_stderr);
1173  2          }

1175  2          /*
1176  2          ** EDMLINK: clean up and shut down the edmlink api
1177  2          */

1179  2          if( NULL != pELinkTargetObj )
1180  2              (void) ELinkDestroyObj( pELinkHandle, pELinkTargetObj );

1182  2          if( NULL != pELinkUserIdObj )
1183  2              (void) ELinkDestroyObj( pELinkHandle, pELinkUserIdObj );

1185  2          if( NULL != pELinkCmdObj )
1186  2              (void) ELinkDestroyObj( pELinkHandle, pELinkCmdObj );

1188  2          (void) ELinkDoneAPI( pELinkHandle );

1190  2          if (0 != ELinkStatus)
1191  3          {
1192  3              sprintf(cxp->ap_error_message,
1193  3                      "cannot set up remote connection when calling "
1194  3                      "ELinkShell with %s, %d, %s, %s, \"%s\",
                                                  error \"%s\" (%d)",
1195  3                      rcmd_stuff[0], cxp->ap_shelltcp_port,
                                                  rcmd_stuff[1],
1196  3                      rcmd_stuff[2], rcmd_stuff[3],
1197  3                      esl_strerror(
                                ELinkStatus_errno), ELinkStatus_errno);

1199  3              pkt0.failcode = 2;
1200  3              pkt0_errstr = cxp->ap_error_message;

1202  3              if (debugmode)
1203  4              {
1204  4                  rbe_log_stats(
```

```
                                ELinkStatus_errno, "ELinkShell failed!\n");
            }

            goto send_0;
        }

        /* add remainder of Symmetrix Path handshaking here
          if the we made it through the other Symm Path tests
        */
        if (symmPathOK == TRUE)
        {
          int newfd;  /* fd for SSL sopen call */

          /*
           * We're going to grab the short name SSL alias for the
           * long network name.
                                Useless unless this is a cross-restore.
           * (
             because rcmd_stuff[0] == pwi->ssl_groupname on a regular
           *  restore)
           */
          memset(sslName, 0, CDL_HOST_LENGTH);
          strncpy(sslName, rcmd_stuff[0], CDL_HOST_LENGTH-1);

          if ((strlen(rcmd_stuff[0]) >= CDL_HOST_LENGTH) &&
              (0 >= CDL_getsslhostname(sslName, rcmd_stuff[0])))
            {
              char errStr[128] ;
              sprintf(
      errStr, "Unable to determine the short SymmPath hostname of %s",
                                                        rcmd_stuff[0]);
              rbe_user_error(0, errStr);
              pkt0.failcode = 2;
              pkt0_errstr = "Cross-restore is configured to go
                             through SP, yet destination host is not"
                " properly configured in edm.conf";

              goto send_0;
            }

          /*
          * Establish the SymmPath channel
          */
         if ((newfd = CDL_client(rcmd_fd[0],
                                 sslName,
                                 sslGroup)) <0)

          {
            rbe_user_error(0,
                           "Unable to open an SSL listener connection
                                                  -- CDL_client failed");
            pkt0.failcode = 2;
            pkt0_errstr = CDL_errstr(newfd);


            goto send_0;
          }

          /* We connected...Use it.. */
          rcmd_fd[0] = rcmd_fd[1] = newfd ;

        }



    }
    else if (0 == strcmp(method, "netware"))
```



```
    {
        /*
         * netware connection method here
         */

        char buf9[EB_MAXPATHLEN];
        char targetbuf[EB_MAXPATHLEN];
        char *tsa = "?";
        char *target = NULL;


        filter_cmd_argv[filter_cmd_argc++] = "-/";
                                                /* always full path */
        filter_cmd_argv[filter_cmd_argc] = NULL;

       /*
        * Set up the rcmd connection (rsh method) to the client.
        */

       if (! cxp->ap_have_shelltcp_port)
        {
           struct servent *sp;

           /*
            * Try to get the port number again.
            */

           if ((sp = getservbyname("shell", "tcp")) != NULL)
            {
               cxp->ap_shelltcp_port = (ushort_t)sp->s_port;
               cxp->ap_have_shelltcp_port = 1;
            }
           else
            {
               pkt0.failcode = 1;
               pkt0_errstr = "shell/tcp service not found";
               goto send_0;
            }
        }

       rcmd_fd = rcmd_fds; /* set valid pointer */

        /*
         * now skip over "./host/bin/startrec"
         */

        for (p2 = rcmd_stuff[3]; '\0' != *p2 && ' ' != *p2; p2++)
        {
            /* skip ./host/bin/startrec */
        }

        while (' ' == *p2)
        {
            p2++; /* advance to start of next word */
        }

        /*
         * if a -c target:/path then take out target:
         */

        *targetbuf = '\0';
        if (strlen(p2) >=2)
          {
            if (('-' == *p2) && ('c' == p2[1]))
              {
                char *p3 = p2+2;
```

```
1329  4                    char *p4;
1330  4                    while (' ' == *p3)
1331  5                      {
1332  5                        p3++; /* advance to start of next word */
1333  4                      }
1334  4                    p4 = p3; /* save this as start of the prefix */
1335  4                    for (;'\0' != *p3; p3++)
1336  5                      {
1337  5                        if ((':' == *p3) && ('/' == p3[1])) ||
1338  5                            ((':' == *p3) && ('\"' == p3[1])))
1339  6                          {
1340  6                            break;
1341  5                          }
1342  4                      }
1343  4                    if (*p3 != 0) /* if found the prefix */
1344  5                      {
1345  5                        char * tempChar = NULL;
1346  5                        strncpy(targetbuf, p4, p3-p4);
1347  5                        targetbuf[p3-p4] = '\0';
1348  5                        target = targetbuf;
1349  5                        if('\"' == p3[1])
1350  6                          {
1351  6                            p3++;
1352  5                          }
1353  5                        tempChar = strchr(p3+1,'\"');
1354  5                        if (NULL != tempChar)
1355  6                          {
1356  6                            tempChar[0] = ' ';
1357  5                          }
1358  5                        strcpy(p4, p3+1);
1359  4                      }
1360  3                  }
1361  2                }

1363  2              /*
1364  2               * now, if it's a netware cross recovery, get password,
                                                                    etc. from
1365  2               * destination target -- don't use source target from pw.
1366  2               * pwi points to work item config struct backup was done
                                                                    under
1367  2               */

1369  2              if (0 != strcmp(
                            rcmd_stuff[0], pwi->sysname)) /* if x-recovery */
1370  3              {
1371  3                  RBC_WORKITEM *pwi2 = pwi;


1374  3                  tsa = "?";
1375  3                  if (*targetbuf != 0)
1376  4                  {
1377  4                      target = targetbuf;
1378  3                  }

1380  3                  for (pwg = cxp->ap_config->pgrouplist; NULL != pwg;
                                                          pwg = pwg->next)
1381  4                  {
1382  4                      for (pwi = pwg->pwilist; NULL != pwi; pwi = pwi->next)
1383  5                      {
1384  5                          if (ebc_same_host(pwi->sysname, rcmd_stuff[0]))
1385  6                          {
1386  6                              goto gotit2;
1387  5                          }
1388  4                      }
1389  3                  }
1390  3  gotit2:
```



```
1391  3                  /*
1392  3                   * if we did not find a wi for the target system,
1393  3                   * put user prompting here.
1394  3                   */

1396  3                  if (NULL == pwi)
1397  4                  {
1398  4                      pwi = pwi2; /* restore old value if ! found */
1399  3                  }
1400  2              }

1402  2              /*
1403  2               * check for an encrypted password
1404  2               */
1405  2              if (NULL != pwi && (
                                pwi->flags & WORKITEM_FLAGS_ENCRYPT_PASSWD))
1406  3              {
1407  3                  sprintf(
                     buf9, "r %s -target %s -tsa %s -login %s -epassword %s", p2,
1408  3                            ((target != NULL)
1409  3                             ? target
1410  3                             : ((
                      pwi->nw_clnt_target != NULL) ? pwi->nw_clnt_target : "?")),
1411  3                            tsa,
1412  3                            ((
                            pwi->nw_username != NULL) ? pwi->nw_username : "?"),
1413  3                            ((
                               pwi->nw_passwd != NULL) ? pwi->nw_passwd : "?"));
1414  2              }
1415  2              else
1416  3              {
1417  3                  sprintf(
                      buf9, "r %s -target %s -tsa %s -login %s -password %s", p2,
1418  3                            ((target != NULL)
1419  3                             ? target
1420  3                             : ((
                      pwi->nw_clnt_target != NULL) ? pwi->nw_clnt_target : "?")),
1421  3                            tsa,
1422  3                            ((
                            pwi->nw_username != NULL) ? pwi->nw_username : "?"),
1423  3                            ((
                               pwi->nw_passwd != NULL) ? pwi->nw_passwd : "?"));
1424  2              }

1426  2              if (NULL != pwi && (CNCTN_RSH != pwi->connection_type))
1427  3              {
1428  3                  if (CNCTN_NETWARE != pwi->connection_type)
1429  4                  {
1430  4                      WriteFmtStringMsg(cxp->ap_w_prog_fd,
1431  4                                        EDMREPROGMSG_AUXPROC_WARNING, 0,
1432  4                                        "work item \"%s\" specifies
                                                connection method %s but client "
1433  4                                        "was installed to use the netware
                                                 method -- using netware method",
1434  4                                        pwi->name,
1435  4                                        ((
                                      CNCTN_RSH == pwi->connection_type) ? "Rsh" :
1436  4                                       ((
                                 CNCTN_EDMLINK == pwi->connection_type) ? "Edmlink" :
1437  4                                        ((
                                   CNCTN_DIRECT == pwi->connection_type) ? "Direct" :
1438  4                                         ((
                                   CNCTN_SOCKET == pwi->connection_type) ? "Socket" :
1439  4                                          ((
                                 CNCTN_NETWARE == pwi->connection_type) ? "Netware" :
1440  4                                           "???"))))));
```

```
1441 3              }
1443 3              if (0 != pwi->connection_port) /* if port specified */
1444 4              {
1445 4                  if (debugmode)
1446 5                  {
1447 5                     rbe_log_stats(
                                  0, "using socket port %d to connect to"
1448 5                                   " host \"%s\", witem \"%s\"",
1449 5                                  pwi->connection_port, pwi->sysname,
                                                              pwi->name);
1450 4                  }
1451 4                  cxp->ap_shelltcp_port = (ushort_t)pwi->connection_port;
1452 3              }
1453 2          }
1455 2          if (debugmode)
1456 3           {
1457 3              rbe_log_stats(
                           0, "Calling nwrcmd with %s, %d, %s, %s, %s\n",
1458 3                          rcmd_stuff[0], cxp->ap_shelltcp_port,
1459 3                          rcmd_stuff[1], rcmd_stuff[2], buf9);
1460 2           }
1462 2          rcmd_fd[0] = nwrcmd(&rcmd_stuff[0], cxp->ap_shelltcp_port,
1463 2                              rcmd_stuff[1], rcmd_stuff[2], buf9,
                                                              &rcmd_stderr,
1464 2                              pwi->ssl_groupname, pwi->ssl_clientname,
                                                                  TRUE);
1465 2          rcmd_fd[1] = rcmd_fd[0]; /* this one is bi-directional */
1467 2          if (debugmode)
1468 3           {
1469 3              rbe_log_stats(
                        0, "AUXPROC: nwrcmd returned fd %d, stderr fd %d\n",
1470 3                          rcmd_fd[0], rcmd_stderr);
1471 2           }
1473 2          if (rcmd_fd[0] == -1)
1474 3           {
1475 3              pkt0.failcode = 2;
1476 3              pkt0_errstr = "cannot set up remote connection";
1478 3               goto send_0;
1479 2           }
1480 1      }
1481 1      else if (0 == strcmp(method, "socket"))
1482 2      {
1483 2           /*
1484 2            * socket connection method here
1485 2            */
1487 2          if (CNCTN_RSH != pwi->connection_type) /* if other than rsh */
1488 3           {
1489 3               if (CNCTN_SOCKET != pwi->connection_type)
1490 4               {
1491 4                  WriteFmtStringMsg(cxp->ap_w_prog_fd,
1492 4                                    EDMREPROGMSG_AUXPROC_WARNING, 0,
1493 4                                    "work item \"%s\" specifies
                                                  connection method %s but "
1494 4                                    "client was installed to use the
                                         socket method -- using socket method",
1495 4                                    pwi->name,
1496 4                                    ((
                                  CNCTN_RSH == pwi->connection_type) ? "Rsh" :
1497 4                                    ((
```



```
                  CNCTN_EDMLINK == pwi->connection_type) ? "Edmlink" :
1498 4                                    ((
                    CNCTN_DIRECT == pwi->connection_type) ? "Direct" :
1499 4                                    ((
                    CNCTN_SOCKET == pwi->connection_type) ? "Socket" :
1500 4                                    ((
               CNCTN_NETWARE == pwi->connection_type) ? "Netware" :
1501 4                                    "???")))))));
1502 3          }
1504 3           /*
1505 3            * read file with name of WI (
                          written by listener) to get info about client
1506 3            */
1508 3          socket_port  = -1;
1509 3          dbseqno = 1;
1510 3          strcpy(wifilename, wifiledir);
1511 3          if (NULL == socket_file)
1512 4            {
1513 4              rbe_log_stats(
                          0, "No socket file name passed to auxproc\n");
1514 4              return;
1515 3            }
1516 3          strcat(wifilename, socket_file);
1518 3          /*
1519 3           * Get socket information from eblistend info file
1520 3           */
1521 3          errnum = eb_parse_listener_info_file (wifilename,
1522 3                                                cbuf,
1523 3                                                socket_host,
1524 3                                                &socket_port,
1525 3                                                &dbseqno,
1526 3                                                listener_filename);
1528 3          if (errnum != E_SUCCESS)
1529 4          {
1530 4              /*
1531 4               * The routine already logged an error message,
                                                                  simply
1532 4               * return the error to caller
1533 4               */
1534 4              rbe_log_stats(0,
1535 4                            "Unable to read eblistend info file
                                                              \"%s\"\n",
1536 4                            wifilename);
1537 4              return;
1538 3          }
1540 3          if (socket_port == -1)
1541 4          {
1542 4              rbe_log_stats(
                          0, "Invalid port field seen for client\n");
1543 4              return;
1544 3          }
1546 3           if (
        0 != pwi->connection_port) /* if port specified in work item */
1547 4          {
1548 4              if (debugmode)
1549 5              {
1550 5                  rbe_log_stats(
                                0, "using socket port %d to connect to"
1551 5                                 " host \"%s\", witem \"%s\"",
1552 5                                pwi->connection_port, pwi->sysname,
```

```
                                                                pwi->name);
1553  4                 }
1554  4                 socket_port = pwi->connection_port;
1555  3              }
1556  2           }

1558  2           if (debugmode)
1559  3           {
1560  3              rbe_log_stats(
      0, "host= %s port=%d seq#=%d\n", socket_host, socket_port, dbseqno);
1561  2           }

1563  2           rcmd_fd = rcmd_fds; /* set valid pointer */

1565  2           /* We need to fix this up...
                                      Modify sopen to call the right type
1566  2                 of socket (STP) */

1568  2           rcmd_fd[0] = sopen(socket_host, 'c', socket_port);
1569  2           if (rcmd_fd[0] < 0)
1570  3           {
1571  3              rbe_log_stats(0, "Socket Open error: %d\n", rcmd_fd[0]);
1572  3              _exit(2);
1573  2           }

1575  2           /*
1576  2            * set up ONLY remote command file descriptors to talk to
                                                                    client
1577  2            * via socket connection. DO NOT overwrite filter_cmd fd's
1578  2            */

1580  2           rcmd_fd[1]  = dup(rcmd_fd[0]);

1582  2           /* If we are connected,
                                determine if the user wants normal sockets
1583  2            * or ssl.
1584  2            */

1586  2           if (rcmd_fd[0] != -1)
1587  3           {
1588  3               if (!pwi->ssl_groupname)
1589  4               {
1590  4                   /*
1591  4                    * send OK status to client waiting on data socket
                                                                      before
1592  4                    * data stream
1593  4                    */

1595  4                   sprintf(cbuf,"OK: %d socket connection made;
                                                             ", dbseqno);
1596  4                   (void)swrite (rcmd_fd[0], cbuf, (int)strlen(cbuf));
1597  3               }
1598  3               else /* ssl connect */
1599  4               {
1600  4                   char sslName[CDL_HOST_LENGTH];
1601  4                   char sslGroup[CDL_STG_LENGTH];
1602  4                   char **stgNames;
                                       /* for returned array of STG names */
1603  4                   int  numGroups = 0;
                                      /* for returned number of STG names */
1604  4                   int  addIndex = 0;
1605  4                   int  rc = 0;

1607  4                   /* First test to see if this is a cross restore. If
1608  4                  * so we can't just use the STG group from the source
                                                                    client
```



```
1609  4                  * since it may not exist on the destination client.
1610  4                  * We need to see if the destination is SP enabled and
1611  4                  * get a good STG group for that.
                                                      We'll use the same group
1612  4                  * if it is available.
                                          If no STG group is available we will
1613  4                  * fall back to network.
1614  4                  */

1616  4                   if (!ebc_same_host(
                              socket_host, pwi->sysname)) /* if x-recovery */
1617  5                   {
1618  5                         /* Grab the STG entries for this client */
1619  5                         rc = CDL_getavailablegroups(
                                        socket_host, &numGroups, &stgNames);
1620  5                         if ((numGroups==0) || (rc < 0))
1621  6                         {
1622  6                           char errStr[128] ;
1623  6                           sprintf(
      errStr, "Unable to use SymmPath for cross restore to host %s--using
                                                             network", socket_host);
1624  6                           rbe_user_error(0, errStr);
1625  6                           /*
1626  6                            * Default to network and send normal
                                                                  handshake.
1627  6                            * send OK status to client waiting on data
                                                               socket before
1628  6                            * data stream
1629  6                            */

1631  6                           sprintf(cbuf,"OK: %d socket connection made;
                                                                   ", dbseqno);
1632  6                           (void)swrite (rcmd_fd[0], cbuf, (int)strlen(
                                                                       cbuf));


1634  5                         }
1635  5                         else
1636  6                         {
1637  6                             /*
1638  6                              * The destination client is SP enabled.
                                                                 First check
1639  6                              * to see if the same STG group exists.
                                                                 If we don't
1640  6                              * find the same one we will use the first,
                                                                    so set
1641  6                              * that as a default
1642  6                              */

1644  6                               memset(sslGroup,0,CDL_STG_LENGTH);
1645  6                               strncpy(
                                           sslGroup,stgNames[0],CDL_STG_LENGTH-1);

1647  6                              /* Initialize the index */
1648  6                              addIndex = numGroups;
1649  6                              while (--addIndex >= 0)
1650  6                                 if (0 == strcmp(
                                           pwi->ssl_groupname, stgNames[addIndex]))
1651  7                                 {
1652  7                                    strncpy(
                                     sslGroup,stgNames[addIndex],CDL_STG_LENGTH-1);
1653  7                                    continue;
1654  6                                 }

1656  6                              /*
1657  6                               * We need to get a short name SSL alias if
                                                                 the destination
```

```
1658 6                          * name happens to be long
1659 6                          */

1661 6                          memset(sslName, 0, CDL_HOST_LENGTH);
1662 6                          strncpy(
                                  sslName, socket_host, CDL_HOST_LENGTH-1);

1664 6                          if ((strlen(
                                        socket_host) >= CDL_HOST_LENGTH) &&
1665 6                                 (0 >= CDL_getsslhostname(
                                                sslName, socket_host)))

1667 7                          {
1668 7                            char errStr[128] ;
1669 7                            sprintf(
        errStr, "Unable to determine the short SymmPath hostname of %s",
                                                                socket_host);
1670 7                            rbe_user_error(ffdb_errnum, errStr);
1671 7                            pkt0.failcode = 2;
1672 7                            pkt0_errstr = "Cross-restore is
              configured to go through SP, yet destination host is not"
1673 7                              " properly configured in edm.conf";

1675 7                            goto send_0;
1676 6                          }
1677 6                          if (ebsock_server_connect_ssl(sslGroup,
1678 6                                                        sslName,
1679 6                                                        &rcmd_fd[0],
                                                             &rcmd_fd[1],
1680 6                                                        socket_host,
1681 6                                                        dbseqno,
1682 6                                                        holdbuff) <
                                                                      0)

1683 7                            {
1684 7                              WriteFmtStringMsg(cxp->ap_w_prog_fd,
1685 7
                                            EDMREPROGMSG_AUXPROC_ERROR, 0,
                                                 /* "%s",*/ holdbuff);
1686 7                              return;
1687 7                            }
1688 6                        }
1689 5                    }
1690 4                    else
1691 4                      {
1692 5                        /*
1693 5                         * Just a normal Sp restore--nothing special to
                                                                        do
1694 5                         */

1695 5                        if (ebsock_server_connect_ssl(
                                                  pwi->ssl_groupname,
1697 5                                               pwi->ssl_clientname,
1698 5                                               &rcmd_fd[0],
1699 5                                                    &rcmd_fd[1],
1700 5                                               socket_host,
1701 5                                               dbseqno,
1702 5                                               holdbuff) < 0)

1703 6                          {
1704 6                            WriteFmtStringMsg(cxp->ap_w_prog_fd,
1705 6                                              EDMREPROGMSG_AUXPROC_ERROR,
                                                                      0,
1706 6                                              /* "%s",*/ holdbuff);
1707 6                            return;
1708 5                          }
1709 4                      }
1710 3                  }
```



```
1711 2          }

1713 2          /* Now that we are finally connected, dup to std err */
1714 2        rcmd_stderr = CDL_dup(rcmd_fd[0]);

1716 1      }
1717 1      else
1718 2      {
1719 2           /*
1720 2            * unknown connection method here
1721 2            */
1722 2        WriteFmtStringMsg(
                              cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
1723 2                            "Bad connection method \"%s\"\n", method);
1724 2        return;
1725 1      }

1727 1      if( 0 != pipe(xcpiogen_pipe))
1728 2      {
1729 2        WriteFmtStringMsg(
                              cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
1730 2                            "Could not set up progress channel from
                                                                  xcpiogen.\n");

1731 2        _exit(2);
1732 1      }

1734 1      switch (xcpiogen_pid = pkt0.pid = fork())
1735 2      {
1736 2      case -1:
1737 2           /* error */
1738 2          (void) close(xcpiogen_pipe[0]);
1739 2          (void) close(xcpiogen_pipe[1]);
1740 2           pkt0_errstr = "fork failed";
1741 2           pkt0.failcode = 3;
1742 2           pkt0.errnum = errno;
1743 2           break;

1745 2      case 0:
1746 2           /* child */

1748 2           /*
1749 2            * Make our stdin be the bulk-from-parent stream.
1750 2            */
1751 2           (void)dup2(cxp->ap_r_bulk_fd, 0);

1753 2           if (debugmode)
1754 3           {
1755 3               rbe_log_stats(
                                0, "FILTER CMD: \"%s\", human_name: \"%s\"\n",
1756 3                                 filter_cmd, human_name);
1757 3               for (i = 0; i < filter_cmd_argc; i++)
1758 4               {
1759 4                   rbe_log_stats(0, "\t%s\n", filter_cmd_argv[i]);
1760 3               }
1761 2           }

1763 2           /*
1764 2            * If no fd info passed to child,
                                                then child stdout goes to the
1765 2            * rcmd process.  Otherwise, arbitrary fd is used,
                                                                and we pass
1766 2            * the fd number as an argument per the format given to us
1767 2            */

1769 2           if (filter_cmd_fd_info == -1)
1770 3           {
```

```
1771  3              filter_cmd_fdout = 1;
1772  3              (void)dup2(rcmd_fd[0], 1);
1773  2          }
1774  2          else
1775  3          {
1776  3              filter_cmd_fdout = rcmd_fd[0];
1777  3              (void) sprintf(
                                fdarg, filter_cmd_argv[filter_cmd_fd_info],
1778  3                               filter_cmd_fdout);
1779  3              filter_cmd_argv[filter_cmd_fd_info] = fdarg;
1780  2          }

1782  2          /*
1783  2           * Send real uid string on filter_cmd_fdout to client process
1784  2           */

1786  2          i = (int)strlen(human_name);
1787  2          if ((n = looprw(
                  filter_cmd_fdout, human_name, i, write_CDL_no_eintr)) != i)
1788  3          {
1789  3              rbe_log_stats(RBRECOVER_MKERR(
                            errno), "Can't write %d bytes to host \"%s\", "
1790  3                              "ec=%d, errno=%d, fd=%d\n", i,rcmd_stuff[0],
                                                                    n, errno,
1791  3                              filter_cmd_fdout);
1792  3              _exit(2);
1793  2          }

1795  2          if ((n = looprw(
                        filter_cmd_fdout, "\n", 1, write_CDL_no_eintr)) != 1)
1796  3          {
1797  3              rbe_log_stats(RBRECOVER_MKERR(errno),
1798  3                              "Can't write %d bytes to host \"%s\", "
1799  3                              "ec=%d, errno=%d, fd=%d\n",
1800  3                              1,rcmd_stuff[0],n,errno,filter_cmd_fdout);
1801  3              _exit(3);
1802  2          }

1804  2          (void)close(xcpiogen_pipe[0]);
1805  2         (void)dup2(xcpiogen_pipe[1], 2);
1806  2          (void)close(xcpiogen_pipe[1]);

1808  2          /* Call the CDL layer so we can warn SSL that an execvp
1809  2          is coming. If this isn't an SSL socket this call is a
1810  2          no-op
1811  2          */
1812  2          (void)CDL_execvpPrep(filter_cmd_fdout);

1814  2          (void)execvp(filter_cmd, filter_cmd_argv); /* run xcpiogen */
1815  2          rbe_log_stats(RBRECOVER_MKERR(
                                errno), "Can't exec \"%s\", errno=%d\n",
1816  2                        filter_cmd,errno);
1817  2          _exit(1);
1818  2          break;




1822  2      default:
1823  2           /* parent */
1824  2        (void)close(xcpiogen_pipe[1]);
1825  2        xcpiogen_prog_fd = xcpiogen_pipe[0];
1826  2        break;
1827  1      }   /* end of switch */

1829  1      /* Parent has no use for this file descriptor any more,
                                                              and nuking it
```



```
1830  1      * now is important so that if the child dies while the rcmd still
1831  1      * wants more input from the child, the rcmd will die too (
                                                              instead of
1832  1      * hanging around hoping that maybe this parent process will
                                                                  supply
1833  1      * the data) */

1835  1     /* we didn't really dup these if they were SFP sockets so
1836  1       avoid the closes in that case
1837  1     */
1838  1     if (!pwi->ssl_groupname)
1839  2     {
1840  2          if (rcmd_fd[0] != -1)
1841  3          {
1842  3             (void)close(rcmd_fd[0]);
1843  2          }
1844  2          if (rcmd_fd[1] != -1)
1845  3          {
1846  3             (void)close(rcmd_fd[1]);
1847  2          }
1848  1     }
1849  1     /* However in the case of nwrcmd we did get a network
1850  1       socket for  rcmd_stderr so we do need to issue one
1851  1       CDL_close on the one SFP socket. We can detect that
1852  1       case because rcmd_stderr will not be equal to rcmd_fd[0]
1853  1     */
1854  1     else
1855  2     {
1856  2          if (rcmd_stderr != rcmd_fd[0])
1857  3          {
1858  3              if (rcmd_fd[0] != -1)
1859  4              {
1860  4               /*
1861  4                * IF not a SSL socket, do the close if not
1862  4                * we close SP socket in XCPIOGEN after
1863  4                * data has been moved by calling STPexit()
1864  4                */

1866  4               if ( 0 == CDL_isSSLsocket(rcmd_fd[0]) )
1867  5               {
1868  5                 (void) CDL_close(rcmd_fd[0]);
1869  4               }

1871  4               rcmd_fd[0] = -1;
1872  3              }
1873  2          }
1874  1     }

1876  1     if ( (0 == strcmp(method, "rsh")) || (0 == strcmp(
                                                        method, "edmlink")) )
1877  2     {
1878  2      /* Because of occasional loss of data, due to timing issues when
1879  2       * connected via edmlink,
                                turn on keepalive.  In version 4.7 this will
1880  2       * also be addressed by edmlink, directly. (OSGsw38596)
1881  2       */
1882  2         int on = 1;
1883  2         if (-1 == setsockopt( rcmd_stderr,
1884  2                               SOL_SOCKET,
1885  2                               SO_KEEPALIVE,
1886  2                               (char *)&on,
1887  2                               sizeof(on) ) )
1888  3         {
1889  3             rbe_log_stats( RBRECOVER_MKERR(errno),
1890  3                            "Warning: setsockopt for SO_KEEPALIVE
                                                                  failed" );
```

```
1891 2          }
1892 1      }

1894 1      if(0 == pkt0.failcode)
1895 2      {
1896 2        /* Lets check here to see if xcpiogen is still running.
1897 2         * The child could have exited prior to the fork.
1898 2         */
1899 2        int ChildDone_ret;
1900 2        int ChildExitStatus;

1902 2        r_resultdata = 0;           /* "meaningless" */

1904 2        xcpiogen_still_running = TRUE;
1905 2        sleep (1);
1906 2        ChildDone_ret = ChildDone(xcpiogen_pid, &ChildExitStatus);
1907 2        if(-1 == ChildDone_ret)
1908 3        {
1909 3           rbe_log_stats(RBRECOVER_MKERR(errno),
1910 3                         "Internal error: testing to see if xcpiogen
                                                                  exited.");

1911 2        }
1912 2        else if (0 != ChildDone_ret)
1913 3        {
1914 3           xcpiogen_still_running = FALSE;
1915 3           r_resultdata = ChildExitStatus;
1916 2        }
1917 1      }
1918 1      /*
1919 1       * Send the setup failure/success packet.
1920 1       */

1922 1  send_0:
1923 1      if (pkt0.msglen < 0)
1924 2      {
1925 2           pkt0.msglen = (int)strlen(pkt0_errstr) + 1;
1926 1      }

1928 1      c = '0';
1929 1      i = (int)sizeof(pkt0) + pkt0.msglen;
1930 1      pwrite_or_die(cxp->ap_w_fd, &c, 1, _exit);
1931 1      pwrite_or_die(cxp->ap_w_fd, (char *)&i, sizeof i, _exit);
1932 1      pwrite_or_die(cxp->ap_w_fd, (char *)&pkt0, sizeof pkt0, _exit);

1934 1      if (pkt0.msglen > 0)
1935 2      {
1936 2           sprintf(cxp->ap_error_message, "%s", pkt0_errstr);
1937 2           rbe_log_stats(0, "%s", cxp->ap_error_message);
1938 2           pwrite_or_die(cxp->ap_w_fd, pkt0_errstr, pkt0.msglen, _exit);
1939 1      }

1941 1      /* If the fork was successful,
                              wait for the remote exit status to come back
1942 1       * on stderr,
                    and send it to parent in an 'R' reply.  If the fork was
1943 1       * unsuccesful, there is no 'R' reply, and, furthermore,
                                                          the value in the
1944 1       * 'r' reply is meaningless. */


1947 1      if ((xcpiogen_pid > 0) &&
1948 1           (TRUE == xcpiogen_still_running))
1949 2      {
1950 2           int remote_exitinfo;
1951 2           int Demux_ret = 0;
1952 2           Demux_ret = DemuxAuxChildren(cxp->ap_w_prog_fd,
```



```
1953 2                                        rcmd_stderr,
1954 2                                        rcmd_stuff[0],
1955 2                                        xcpiogen_prog_fd,
1956 2                                        xcpiogen_pid,
1957 2                                        &remote_exitinfo);
1958 2           if(0 != Demux_ret)
1959 3           {
1960 3             /*      rbe_log_stats(
                                0, "Error monitoring Auxproc's children.");*/
1961 3             /* error_logging */
1962 2           }


1965 2           /*
1966 2            * notify the main program of remote success/failure
1967 2            */

1969 2           c = 'R';
1970 2           i = sizeof(remote_exitinfo);
1971 2           pwrite_or_die(cxp->ap_w_fd, &c, 1, _exit);
1972 2           pwrite_or_die(cxp->ap_w_fd, (char *)&i, sizeof i, _exit);
1973 2           pwrite_or_die(cxp->ap_w_fd, (
                                        char *)&remote_exitinfo, i, _exit);

1974 2  #if 0
1975 2           rbe_log_stats(0, "Auxproc(%d) remote exit is %d.", getpid(),
1976 2                         remote_exitinfo);
1977 2  #endif
1978 2           /*
1979 2            * Is our connection method anything BUT "netware"?
1980 2            */

1982 2           if (0 != strcmp(method, "netware"))
1983 3           {      /***  YES  ***/
1984 3               /* Now wait for the exit code of the filter (
                                                              local) process
1985 3                *
1986 3                * waitpid() may return with EINTR before the child
1987 3                * proc exits, in that case, we want to continue
1988 3                * with the wait. (fix for OSGsw15487)
1989 3                */

1991 3               while ((waitpid(xcpiogen_pid, &wait_result, 0) == -1) &&
1992 3                      (EINTR == errno))
1993 4               {
1994 4                   /* empty while loop */
1995 3               }

1997 3               /*
1998 3                * exited while loop either because the return value of
1999 3                * waitpid is NOT -1, or the errno is NOT EINTR
2000 3                */
2001 2           }
2002 2           else
2003 3           {      /*** IT'S NETWARE ***/
2004 3               /*
2005 3                *
2006 3                ************ HACK **** HACK **** HACK **** HACK
                                                              ************
2007 3                ************ HACK **** HACK **** HACK **** HACK
                                                              ************
2008 3                ************ HACK **** HACK **** HACK **** HACK
                                                              ************
2009 3                ************ HACK **** HACK **** HACK **** HACK
                                                              ************
2010 3                ************ HACK **** HACK **** HACK **** HACK
                                                              ************
```

```
2011  3                *
2012  3                *  This hack is basically for Netware.
                                                    Netware TCP/IP close()
2013  3                *  does not send anything back to the (
                                                      local)filter process
2014  3                *  that is running as pid pkt0.pid.
                                                   So the "filter" process
2015  3                *  just keeps on sending data across the network and
                                                          there isn't
2016  3                *  anybody there to read it.
2017  3                *
2018  3                *  Therefore,
                              this "auxproc" doesn't finish until all data has
2019  3                *  been written to the socket even though no process is
                                                                  reading
2020  3                *  it. However,
                                 the process might hang indefinitely if the network
2021  3                *  buffer(s) become(s) full.
2022  3                *
2023  3                *  This is how we will get around this (
                                                 kind of a hack but could
2024  3                *  not think of a better way).
                                                 None of the normal TCP/IP
2025  3                *  mechanisms seem to work with Netware.
2026  3                *
2027  3                *  If the exitinfo (
                                  which is zero for success or non-zero for
2028  3                *  failure) reports failure...
2029  3                *
2030  3                *  After we have received the exit info from the Netware
                                                                  client
2031  3                *  then try to get it's "wait_result" twice with a 2
                                                         second pause
2032  3                *  in between.  If we don't get it,
                                                    we have to assume that the
2033  3                *  process is hung and can't wind down.
2034  3                *
2035  3                *  Therefore,
                                 let's send a SIGPIPE to the process.  I chose
2036  3                *  SIGPIPE because this is usually trapped by our
                                                         software and
2037  3                *  handled in a controlled manner.
                                              Just keep on trying to kill
2038  3                *  it, sleeping for 2 seconds,
                                         and then see if it went away until
2039  3                *  it DOES go away.
2040  3                *
2041  3                *  Just keep on trying to kill it,
                                               sleeping for 2 seconds, and
2042  3                *  then see if it went away until it DOES go away.
2043  3                *
2044  3                *  Appearently, the kill(
                                  SIGPIPE) is ignored by the process while
2045  3                *  there is activity occuring like positioning the tape.
                                                                  That is
2046  3                *  the reason for having to do the kill(
                                                       ) in a loop until it
2047  3                *  works.
2048  3                */

2050  3               if (remote_exitinfo)
2051  4               {
2052  4                   /**
2053  4                   ** We recieved an error exit status from Novell
2054  4                   **/
2055  4                   int exitinfo;
```



```
2057  4                   exitinfo = waitpid(
                                        xcpiogen_pid, &wait_result, WNOHANG);
2058  4                   while (!exitinfo)
2059  5                   {
2060  5                       /*
2061  5                        * Wait a couple of seconds longer and try again
2062  5                        */

2064  5                       sleep(2);
2065  5                       exitinfo = waitpid(
                                        xcpiogen_pid, &wait_result, WNOHANG);

2067  5                       if (!exitinfo)
2068  6                       {
2069  6                           kill(xcpiogen_pid, SIGPIPE);
                                                          /*  BLAST IT! */
2070  5                       }
2071  4                   }
2072  3               }
2073  3               else
2074  4               {
2075  4                   /**
2076  4                   ** We received a success return status from Novell
2077  4                   **/

2079  4                   /*
2080  4                    * waitpid() may return with EINTR before the child
2081  4                    * proc exits, in that case, we want to continue
2082  4                    * with the wait. (fix for OSGsw15487)
2083  4                    */

2085  4                   while ((waitpid(
                                    xcpiogen_pid, &wait_result, 0) == -1) &&
2086  4                           (EINTR == errno))
2087  5                   {
2088  5                       /* empty while loop */
2089  4                   }
2090  4                   /*
2091  4                    * exited while loop either because the return value
                                                                        of
2092  4                    * waitpidis NOT -1, or the errno is NOT EINTR
2093  4                    */
2094  3               }
2095  2           }

2097  2           /*
2098  2            * main command loop in caller will send this result for us
2099  2            */

2101  2           /*
2102  2            * wait_result contains the exit status of the child
2103  2            * process. Use the standard macros to determine the
2104  2            * status of the child process and set the return value
2105  2            * to indicate the status accurately.
2106  2            */

2108  2           if (WIFEXITED(
                            wait_result))     /* child proc exited normally */
2109  3           {
2110  3               /*
2111  3                * set the result to the exit code
2112  3                */

2114  3               wait_result = WEXITSTATUS(wait_result);
2115  2           }
```

```
2116 2          else if (WIFSIGNALED(wait_result))
2117 3          {
2118 3              /*
2119 3               * child proc exited due to an uncaught signal, so
2120 3               * set the exit code to the signal number plus our own
2121 3               * code XG_EXIT_SIGBASE.
2122 3               */

2124 3              wait_result = WTERMSIG(wait_result) + XG_EXIT_SIGBASE;
2125 2          }
2126 2          else if (WIFSTOPPED(wait_result))
2127 3          {
2128 3              /*
2129 3               * child proc is stopped, set the exit code accordingly.
2130 3               */

2132 3              wait_result = XG_EXIT_STOPPED;
2133 2          }

2135 2          r_resultdata = wait_result;
2136 1      }

2138 1      if (rcmd_stderr != -1)
2139 2        {
2140 2          /* OSGsw34952 -- Workaround for STPclose() behavioral defect
2141 2           *
2142 2           * Shut off the SSL (STP) exit handler
2143 2           * IF not a SSL socket, do the close if not
2144 2           * we close SP socket in XCPIOGEN after
2145 2           * data has been moved by calling STPexit()
2146 2           */
2147 2          if ( 0 == CDL_isSSLsocket(rcmd_stderr) )
2148 3          {
2149 3            (void)close(rcmd_stderr);
2150 3            rcmd_stderr = -1;
2151 2          }
2152 2          else /* SSL socket used as stderr, so clean up
2153 2                  the SP stuff */
2154 3          {
2155 3            /* SP socket was set to stderr of remote
2156 3               commands.  Since, we call CDL_noatexit
2157 3               when we created the SP socket we will
2158 3               explicitely call CDL_exit() that will
2159 3               clean up the entire SP environment */

2161 3             if ( 0 == sp_cdlexitdone )
2162 4             {
2163 4               CDL_exit();
2164 4               sp_cdlexitdone = 1;
2165 3             }
2166 2          }
2167 1        }

2169 1      /*
2170 1       * set up variables that main loop will use to send 'r' reply
2171 1       */

2173 1      cxp->ap_resultlen = sizeof r_resultdata;
2174 1      cxp->ap_resultdata = (char *)&r_resultdata;
2175    }         /* end of z_rcmdfilter() */
```



```
2178    static enum input_states decodecookie(char *cp);


2181 1  struct cookie2state {
2182 1          char *cookie;
2183 1          enum input_states state;
2184    };

2186 1  static struct cookie2state c2stbl[] = {
2187 1          { REMFD_COOKIE_FD2OUT,          INSTATE_COPY_TO_STDOUT },
2188 1          { REMFD_COOKIE_FD2OUT2,
                                                INSTATE_COPY_TO_STDOUT },
2189 1          { REMFD_COOKIE_FD2OUTEND,       INSTATE_SEARCH_PREFIX0 },
2190 1          { REMFD_COOKIE_STATUS,          INSTATE_GATHER_STATUS },
2191 1          { NULL,                                 INSTATE_NG }
2192    };


2195    static enum input_states
2196    decodecookie(char *cp)
2197 1  {
2198 1      struct cookie2state *c2s;


2201 1      for (c2s = c2stbl; c2s->cookie != NULL; c2s++)
2202 2      {
2203 2          if (strcmp(c2s->cookie, cp) == 0)
2204 3          {
2205 3              return c2s->state;
2206 2          }
2207 1      }
2208 1      return INSTATE_SEARCH_PREFIX0;
2209    }         /* end of decodecookie() */
```

```
2211    static void
2212    sigusr1_handler(int sigvalue)
2213  1 {
2214  1     if (debugmode)
2215  2     {
2216  2       rbe_log_stats(0,"\nUSR1 (ATTN) signal received!\n");
2217  1     }

2219  1     if (0 < xcpiogen_pid)
2220  2     {
2221  2       (void)kill(xcpiogen_pid, SIGTERM);
2222  1     }

2224  1     ++attn_ec;
2225    }         /* end of sigusr1_handler() */
```



```
2227    static void
2228    sigterm_handler(int sigvalue)
2229  1 {
2230  1     if (debugmode)
2231  2     {
2232  2       rbe_log_stats(0,"\nTERM signal received!\n");
2233  1     }

2235  1     rbe_close_logs(logging_channel);

2237  1     /* ESS workaround: Make a call to CDL_exit
2238  1        if symmpath workitem and CDL_exit has
2239  1        not already been called.  This is to clean
2240  1        up sockets since we have made a call to
2241  1        CDL_noatexit, we have to make sure we
2242  1        clean up */

2244  1     if ( (1==is_symmpath) && (0==sp_cdlexitdone) )
2245  2     {
2246  2       CDL_exit();
2247  2       sp_cdlexitdone = 1;
2248  1     }

2250  1     signal(SIGTERM, SIG_DFL); /* restore default (terminate) action */
2251  1     (void)kill(getpid(), SIGTERM);

2253    } /* end of sigattn_handler() */
```

```
2256  /*
2257   * Invoked when running recover in debugging mode.
2258   * This execs a separate a.out file to implement the auxproc, so that
2259   * breakpoints can be set in the recover code without affecting
2260   * the auxproc code.
2261   */

2263  static void
2264  z_exec_separate_auxproc(struct auxproc_context *cxp,
2265                          char *pathname)
2266  1 {
2267  1     char procnum_str[32];
2268  1     char r_fd_str[32];
2269  1     char w_fd_str[32];
2270  1     char r_bulk_fd_str[32];
2271  1     char dbgmodestr[32];
2272  1     char *argv0;

2274  1     /*
2275  1      * NOTE: This name is "special"; the recover main()
2276  1      * function looks for it to know when it's being invoked
2277  1      * to perform auxproc processing.
2278  1      */

2280  1     argv0 = "ebr_auxproc";

2282  1     (void)sprintf(procnum_str, "%d", cxp->ap_my_auxnum);
2283  1     (void)sprintf(r_fd_str, "%d", cxp->ap_r_fd);
2284  1     (void)sprintf(w_fd_str, "%d", cxp->ap_w_fd);
2285  1     (void)sprintf(r_bulk_fd_str, "%d", cxp->ap_r_bulk_fd);
2286  1     (void)sprintf(dbgmodestr, "%d", debugmode);
2287  1     (void)execlp(pathname,              /* prog to execute */
2288  1                  argv0,                 /* argv 0 */
2289  1                  procnum_str,           /* argv 1 */
2290  1                  r_fd_str,              /* argv 2 */
2291  1                  w_fd_str,              /* argv 3 */
2292  1                  r_bulk_fd_str,         /* argv 4 */
2293  1                  dbgmodestr,            /* argv 5 */
2294  1                  (char *)0);            /* end-of-args */

2296  1     /*
2297  1      * if we get here, the exec did not go. Caller will bomb for us.
2298  1      */

2300  }         /* end of z_exec_separate_auxproc() */
```



```
2303  /*
2304   * Use these functions with looprw to build
2305   * a loop read (or loop write) that ignores EINTR.
2306   */

2308  static int
2309  read_CDL_no_eintr(int fd,
2310                    char *buf,
2311                    int nbytes)
2312  1 {
2313  1     int r;

2315  1     do
2316  2     {
2317  2         errno = 0;
2318  2         r = CDL_read(fd, buf, (uint_t)nbytes,0);
2319  1     } while (r == -1 && errno == EINTR);

2321  1     return r;
2322  }         /* end of read_CDL_no_eintr() */
```

```
2324    static int
2325    write_CDL_no_eintr(int fd,
2326                       char *buf,
2327                       int nbytes)
2328  1 {
2329  1     int r;
2330
2331  1     do
2332  2     {
2333  2         errno = 0;
2334  2         r = CDL_write(fd, buf, (uint_t)nbytes,0);
2335  1     } while (r == -1 && errno == EINTR);
2336
2337  1     return r;
2338    }       /* end of write_CDL_no_eintr() */
```



```
2342    /*
2343     * Wait, interruptibly,
                              for at least one byte to become available on fd.
2344     *
2345     * Returns 0 if at least one byte is available.
2346     * Returns -1 for any type of failure, including wait interruption.
2347     * Sets errno appropriate when -1 is returned.
2348     */
2349
2350    int
2351    fd_avail_1_wait_intr(int fd)
2352  1 {
2353  1     esl_fdset_ty rdbits;
2354
2355
2356  1     E_FD_ZERO(&rdbits);
2357  1     E_FD_SET(fd, &rdbits);
2358
2359  1     /*
2360  1      * Don't need to examine rdbits after select, since only one
2361  1      * fd is in the set -- therefore return value can be computed
2362  1      * directly from select return value.
2363  1      */
2364
2365  1     if (esl_select(E_FD_SETSIZE, &rdbits, NULL, NULL, NULL) == -1)
2366  2     {
2367  2         return -1;
2368  1     }
2369  1     return 0;
2370    }       /* end of fd_avail_1_wait_intr() */
```

```
2372  /*
2373   * Test, interruptibly,
                          for at least one byte to become available on fd.
2374   *
2375   * Returns 1 if at least one byte is available.
2376   * Returns -1 for any type of failure, including test interruption.
2377   * Return 0 if no data is available.
2378   *
2379   * Sets errno appropriate when -1 is returned.
2380   */

2382  int
2383  fd_avail_test_intr(int fd)
2384  1 {

2386  1     int retStatus;
2387  1     struct pollfd fd_test;

2389  1     fd_test.fd = fd;
2390  1     fd_test.events = POLLIN;
2391  1     fd_test.revents = 0;


2394  1     /*
2395  1      * Don't need to examine rdbits after select, since only one
2396  1      * fd is in the set -- therefore return value can be computed
2397  1      * directly from select return value.
2398  1      */

2400  1     if ((retStatus = CDL_poll_read(&fd_test, 1, 0)) == -1)
2401  2     {
2402  2       /* ERROR encountered */
2403  2       return -1;
2404  1     }
2405  1     return retStatus;

2407  }         /* end of fd_avail_test_intr() */
```



```
2409  /*
2410   * Test, for at least one byte to become available on fd.
2411   *
2412   * Returns 1 if at least one byte is available.
2413   * Returns -1 for any type of failure other than EINTR.
2414   * Return 0 if no data is available.
2415   *
2416   * Sets errno appropriate when -1 is returned.
2417   */

2419  int fd_avail_test1(int fd)
2420  1 {
2421  1     int retStatus;
2422  1     struct pollfd fd_test;

2424  1     fd_test.fd = fd;
2425  1     fd_test.events = POLLIN;
2426  1     fd_test.revents = 0;


2429  1     /*
2430  1      * Don't need to examine rdbits after select, since only one
2431  1      * fd is in the set -- therefore return value can be computed
2432  1      * directly from select return value.
2433  1      */

2435  1     while ((-1 == (retStatus = CDL_poll_read(&fd_test, 1, 0))) &&
2436  1             (EINTR == errno))
2437  2     {
2438  2       ;
2439  1     }
2440  1     return retStatus;
2441  }
```

```
2443  #define EBR_FORK fork    /* portability definition */
2445  static int ebr_direct_rcmd_fds[2] = { -1, -1 };
2447  /*
2448   * Function to do a direct fork()/exec(
                                    ) instead of an rcmd to start up the
2449   * client programs.
              Returns ptr to in & out fds for stdi/o with client process.
2450   * Returns ptr to fds if successful,
                                      NULL if an error.  stderr from new
2451   * process is returned on *fd2p.
                             Should only be called when client == server.
2452   */
2454  int *
2455  ebr_direct_rcmd(char            **ahost,
2456                  ushort_t          inport,
2457                  struct auxproc_context *cxp,
2458                  char              *locuser,
2459                  char              *remuser,
2460                  char              *cmd,
2461                  int               *fd2p)
2462 1 {
2463 1     int pid;
2464 1     int x[2];
2465 1     int y[2];
2466 1     int z[2];
2467 1     int fd_w;
2468 1     int fd_r;
2469 1     int fd_err;
2470 1     char *client_home = NULL;
2471 1     char *p;
2472 1     struct passwd *pw;
2473 1     struct sigaction    new_act;
2474 1     struct sigaction    old_act;
2475 1     int                 istat;
2478 1     /*
2479 1      * now open pipes that will become daemons stdin, stdout, stderr
2480 1      * remember that x[0] is for read and x[1] is for write
2481 1      */
2483 1     if ((-1 == pipe(x)) || (-1 == pipe(y)) || (-1 == pipe(z)))
2484 2     {
2485 2       WriteFmtStringMsg(
                        cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
2486 2                       "Unable to create a pipe\n");
2487 2       return NULL;
2488 1     }
2489 1     fd_w   = x[1];      /* pipe for child's stdin */
2490 1     fd_r   = y[0];      /* pipe for child's stdout */
2491 1     fd_err = z[0];      /* pipe for child's stderr */
2493 1     /*
2494 1      * Prepare sigaction parameters
2495 1      * ignore SIGCHLD during this
2496 1      */
2498 1     sigemptyset(&new_act.sa_mask);
2499 1     new_act.sa_handler = SIG_IGN;
2500 1     new_act.sa_flags   = E_SA_RESTART;
2501 1     istat = sigaction(SIGCHLD, &new_act, &old_act);
```



```
2503 1     if ((pid = EBR_FORK()) == 0)
2504 2     {
2505 2         /*
2506 2           * child processing goes here while parent is stopped (
                                                        if vfork())
2507 2           * first, setup stdio to work using the pipes
2508 2           */
2510 2         if ((close(0)<0) || (close(1)<0) || (close(2)<0))
2511 3          {
2512 3            WriteFmtStringMsg(cxp->ap_w_prog_fd,
2513 3                              EDMREPROGMSG_AUXPROC_ERROR, 0,
2514 3                              "unable to close std{
                                           in|out|err} for forked child\n");
2515 3            _exit(-1);
2516 2          }
2517 2         if ((dup(x[0])<0) || (dup(y[1])<0) || (dup(z[1])<0))
2518 3          {
2519 3            WriteFmtStringMsg(
                          cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
2520 3                              "unable to dup pipe ends for forked
                                                              child\n");
2521 3            _exit(-1);
2522 2          }
2523 2         if ((close(x[0])<0) || (close(y[0])<0) || (close(z[0])<0)
2524 2               || (close(x[1])<0) || (close(y[1])<0) || (close(z[1])<0))
2525 3          {
2527 3            WriteFmtStringMsg(
                          cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
2528 3                              "unable to close pipe ends for forked
                                                              child\n");
2529 3            _exit(-1);
2530 2          }
2532 2         /*
2533 2          * At this moment the process has a real UID of the user
                                                              executing
2534 2          * [x]ebrecover and an effective UID of root (
                                                    since [x]ebrecover has
2535 2          * the setuid bit set and is owned by root).
2536 2          *
2537 2          * The first problem is the access(
                                          ) function call uses the real UID
2538 2          * of the process.  So if the user is not root or ebadmin,
                                                              the access
2539 2          * function will fail.  This causes the recovery to fail.
                                                                  Thus
2540 2          * problem 1.
2541 2          *
2542 2          * The second problem is System V Unix does not pass the
                                                           effective UID
2543 2          * to processes doing an exec of a shell.
                                                    Instead the effective UID
2544 2          * is changed back to the real UID of the process. However,
                                                                  BSD
2545 2          * implementation does maintain the effective UID of the
                                                              process.
2546 2           *
2547 2          * On System V,
                       neither the real or effective UID will be root during
2548 2          * the exec of the shell script if the user executing
                                                        [x]ebrecover is
2549 2          * a non-root user.  This process will not be able to perform
2550 2          * /bin/sh -c .
                          /<nodename>/bin/startrec because this processes real
```

```
2551 2           * and effective UID (which are now the same) does not have
2552 2           * permissions to the directory ~ebadmin.
2553 2           *
2554 2           * The recxcpio process will be reading the username of the
                                                                  initiating
2555 2           * process from the standard input and setting the real UID to
                                                                      the
2556 2           * UID of that user so it will do no harm to set the real UID
                                                                   to root
2557 2           * at this point.
                          It will be changed back to the actual user as one
2558 2           * of the first things in recxcpio.
2559 2           *
2560 2           */

2562 2          (void)setuid(geteuid());

2564 2          /*
2565 2           * now chdir to the client code home location
2566 2           */

2568 2         if (NULL == (pw = getpwnam(
                                  remuser))) /* lookup client home dir */
2569 3          {
2570 3             p = "Can't get home directory";
2571 3             goto default_home;
2572 2          }
2573 2         else
2574 3          {
2575 3             if ((
          client_home = pw->pw_dir) != NULL) /* if there is a pointer */
2576 4               {
2577 4                  if (0 != chdir(client_home))  /* cd to directory */
2578 5                   {
2579 5                      p = "Can't chdir to home";
2580 5                      goto default_home;
2581 4                   }
2582 3               }
2583 3               else
2584 4               {
2585 4                   p = "Can't figure out home dir";
2586 4   default_home:
2587 4                   WriteFmtStringMsg(cxp->ap_w_prog_fd,
2588 4                                     EDMREPROGMSG_AUXPROC_WARNING, 0,
2589 4                                     "%s;
                      \"%s\" for user \"%s\", defaulting to /usr/epoch/EB\n",
2590 4                                     p, (
                       client_home == NULL) ? "??" : client_home, remuser);
2591 4                   client_home = "/usr/epoch/EB/CLIENT_HOME";
2592 3               }
2593 2          }

2595 2         /*
2596 2           * client_home now points to the home dir of the client
                                                                    software
2597 2           */

2599 2          /*
2600 2           *  Make sure that the home directory that we finally ended
                                                                    up with
2601 2           *  is really there.
2602 2           *
2603 2           *  Yea.. I know.
                        I may have already done this if I found a passwd
2604 2           *  entry for the client backup username and it had a home
                                                                  directory.
```



```
2605 2           *  However,
                       I have to do it again just in case I came from another
2606 2           *  path.
                        I did not really feel like trying to fix up the spagetti
2607 2           *  code.  I'm feeling a little lazy today...
2608 2           */

2610 2          if (0 != chdir(client_home))
2611 3          {
2612 3            WriteFmtStringMsg(
                            cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
2613 3                              "Server backup directory \"%s\" not found
                                                                 or not searchable\n",
2614 3                              client_home);
2615 3            _exit(-1);
2616 2          }

2618 2         (void)execl("/bin/sh", "sh", "-c", cmd, 0);
2619 2          WriteFmtStringMsg(
                            cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
2620 2                            "Unable to execl /bin/sh sh -c %s\n",cmd);
2621 2        _exit(127);
2622 1     }

2624 1     /*
2625 1      * parent code resumes here
2626 1      */

2628 1     /*
2629 1      * Close the parent's copies of the child-ends of the pipes
2630 1      */

2632 1     close(x[0]);                          /* pipe for child's stdin  */
2633 1     close(y[1]);                          /* pipe for child's stdout */
2634 1     close(z[1]);                          /* pipe for child's stderr */

2636 1     /*
2637 1      * check for fork() failure
2638 1      */

2640 1     if (-1 == pid)
2641 2     {
2642 2          WriteFmtStringMsg(
                            cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
2643 2                            "fork() failed\n");

2645 2         close(x[1]);                       /* pipe for child's stdin  */
2646 2         close(y[0]);                       /* pipe for child's stdout */
2647 2         close(z[0]);                       /* pipe for child's stderr */
2648 2          if (istat == 0)
2649 3          {
2650 3              sigaction(SIGCHLD, &old_act, NULL);
2651 2          }
2652 2         return NULL;
2653 1     }

2655 1     /*
2656 1      * now fill in fd's to be returned
2657 1      */

2659 1     ebr_direct_rcmd_fds[0] = fd_w;       /* pipe for child's stdin */
2660 1     ebr_direct_rcmd_fds[1] = fd_r;       /* pipe for child's stdout */
2661 1     if (NULL != fd2p)
2662 2     {
2663 2          *fd2p = fd_err;
2664 1     }
```

```
2666  1      if (istat == 0)
2667  2      {
2668  2           sigaction(SIGCHLD, &old_act, NULL);
2669  1      }

2671  1      return ebr_direct_rcmd_fds;
2672    }    /* end of ebr_direct_rcmd() */
```



```
2674    /*
2675     * routine to search the clients_installed file for a clients
                                                                    connection
2676     * method.  Returns a pointer to the connection method string if
                                                                    successful,
2677     * NULL if the client entry could not be found.
2678     */

2680    static char *
2681    rb_getmethod(register char *host,
2682                 uint_t *concurrency,
2683                 uint_t *client_type)
2684  1 {
2685  1      FILE                  *file_ptr;
2686  1      FILE                  *lock_ptr;
2687  1      FFDB_HOLDER           *inclient_holder;
2688  1      ffdb_inclient_ty      *installed_client;
2689  1      eperrno               errnum;
2690  1      static char           rb_ci_lastmethod[32] = "";

2692  1 #ifdef PARAM_CHECK
2693  1      if (NULL == host)
2694  2      {
2695  2          rbe_internal_error(RBRECOVER_MKERR(
                                                EINVAL), null_arg, "host name");
2696  2          return NULL;
2697  1      }
2698  1 #endif /* PARAM_CHECK */

2700  1      /*
2701  1       * get pathname to file
2702  1       */

2704  1      if (ffdb_inclient_path == NULL) /* if file name needs to be
                                                                    prepared */
2705  2      {
2706  2           if (ffdb_inclient_init(FFDB_INIT_DEFAULT_PATH) != 0)
2707  3           {
2708  3               return NULL;
2709  2           }
2710  1      }

2712  1      /*
2713  1       * request was not in cache -- go load it
2714  1       */

2716  1      if ((lock_ptr = ffdb_lock_file(ffdb_inclient_path,
2717  1                                     "restore scanning
                                                clients_installed file",
2718  1                                     1, NULL, 0)) == NULL)
2719  2      {
2720  2           (void)rbe_user_error(
                      ffdb_errnum, "Unable to obtain lock on \"%s\" for scanning",
2721  2                                ffdb_inclient_path);
2722  2           return NULL;
2723  1      }

2725  1      if ((file_ptr = ffdb_open_file(ffdb_inclient_path)) == NULL)
2726  2      {
2727  2           (void)rbe_user_error(
                          ffdb_errnum, "Unable to open file \"%s\" for scanning",
2728  2                                ffdb_inclient_path);
2729  2           ffdb_unlock_file(lock_ptr);
2730  2           return NULL;
2731  1      }
```

```
2733 1      while ((inclient_holder = ffdb_read_record(
                                            file_ptr, &errnum)) != NULL)
2734 2      {
2735 2          installed_client = (ffdb_inclient_ty *)inclient_holder->drec;
2736 2          if (errnum == 0)
2737 3          {
2738 3              if (ebc_same_host(
                            host, installed_client->hostname)) /* a match? */
2739 4              {
2740 4                  if (concurrency != NULL)
2741 5                  {
2742 5                      if (0 == installed_client->concurrency)
2743 6                      {
2744 6                          *concurrency = 32767;
2745 5                      }
2746 5                      else
2747 6                      {
2748 6                          *concurrency = installed_client->concurrency;
2749 5                      }
2750 4                  }
2751 4                  if (client_type != NULL)
2752 5                  {
2753 5                      *client_type = installed_client->platform;
2754 4                  }
2755 4                  strcpy(rb_ci_lastmethod, installed_client->method);
2756 4                  (void)ffdb_close_file(file_ptr);
2757 4                  ffdb_unlock_file(lock_ptr);
2758 4                  ffdb_destroy_holder(inclient_holder);
2759 4                  return rb_ci_lastmethod;
                                            /* return allocated string */
2760 3              }
2761 2          }
2762 2          ffdb_destroy_holder(inclient_holder);
2763 1      }

2765 1      /*
2766 1       * if we get here -- did not find host in clients_installed file
2767 1       */

2769 1      (void)ffdb_close_file(file_ptr);
2770 1      ffdb_unlock_file(lock_ptr);
2771 1      return NULL;
2772    }   /* end of rb_getmethod() */
```



```
2775    /*
2776     *  Reads remote file descriptor for reads of stderr
2777     *  of the remote cmd. Read is done until nothing is left
2778     *  then returns with or without exit code.
2779     *  The error/warning messages come first and a dealt with,
2780     *  if the exit code of the remote cmd is returned.
2781     *
2782     * (Inp) int fd -- The file descriptor to wait on and read.
2783     * (Out) int *exitp -- The Exit code.
2784     * (Inp) char *remhostname -- The remote client name.
2785     * (Inp) boolean_ty first_call -- Is this the first call ??
2786     * (I/O) enum input_states *state_ptr
2787     * (I/O) enum input_states *next_state_ptr
2788     * (I/O) boolean_ty *skipping_leading_whitespace.
2789     * (I/O) int *parsePos the parse position of the messages.
2790     * (I/O) int *msgPos the position of the logging message.
2791     *
2792     * None of the I/O vars need to be initialized for the first call.
2793     *
2794     * Returns: boolean_ty
2795     * False: The remote command has not exitted.
2796     * True: The remote command has exitted.
2797     */
2798    static boolean_ty
2799    parse_remote_stderr_info2(int prog_fd,
2800                              int remote_fd,
2801                              int *exitp,
2802                              char *remhostname,
2803                              boolean_ty first_call,
2804                              enum input_states *state_ptr,
2805                              enum input_states *next_state_ptr,
2806                              boolean_ty *skipping_leading_whitespace,
2807                              int *parsePos,
2808                              int *msgPos)

2810 1  {
2811 1      enum input_states   previous_state;
2812 1      static char         cookiebuf[REMFD_MAX_COOKIE_LEN+1];
2813 1      int                protfailed          = 0;
2814 1      int                done                = 0;
2815 1      char               c                    = 0;
2816 1      int                n;
2817 1  #define MSGBUFFLEN 260
2818 1      static char         msgLogBuff[MSGBUFFLEN + 1];
2819 1      boolean_ty         ret_status           = FALSE;
2820 1      static int         temp_exit_status;
2821 1      int write_prog = 1;

2823 1      *exitp = 0; /* Initialize to success. */

2825 1      if (TRUE == first_call)
2826 2      {
2827 2        /* First time processing. */

2829 2        if (debugmode)
2830 3        {
2831 3           rbe_log_stats(
                          0, "AUXPROC: Processing stderr info from fd %d\n",
2832 3                        remote_fd);
2833 2        }

2835 2        *state_ptr = INSTATE_NG;
2836 2        *next_state_ptr = INSTATE_SEARCH_PREFIX0;
2837 2        *parsePos = 0;
```

```
2838  2        *exitp = 0;
2839  2        *msgPos = 0;
2840  2        temp_exit_status = 0;
2841  2        *skipping_leading_whitespace = FALSE;
2842  2        n = 0;
2843  1      }
2844  1      else
2845  2      {
2846  2        n = *parsePos;
2847  1      }

2849  1      while (!protfailed && !done)
2850  2      {
2851  2           int start_ec;
2852  2           int r;


2855  2           /*
2856  2            * Wait for the next character from the remote
2857  2            * stderr stream.  Ignore interrupts, unless
2858  2            * they were due to the attention signal.
2859  2            */

2861  2           start_ec = attn_ec;
2862  2           do
2863  3           {
2864  3               r = fd_avail_test_intr(remote_fd);
2865  3               if (0 == r)
2866  4               {
2867  4                 return ret_status;
2868  3               }
2869  2           } while (r == -1 && errno == EINTR && attn_ec == start_ec);

2871  2           /*
2872  2            * now that there is a character, read it
2873  2            */
2874  2           c  = 0;

2876  2           if (r != -1)
2877  3           {
2878  3             do
2879  4             {
2880  4               r = CDL_read(remote_fd, &c, 1, 0);
2881  3             } while((-1 == r) && (errno == EINTR) && (
                                                attn_ec == start_ec));
2882  2           }

2884  2           if (r != 1)
2885  3           {
2886  3               *exitp = SPEXIT_REMOTE_STDERR_FAIL;
2887  3               protfailed = 1;
2888  3               ret_status = TRUE;
2889  3               break;
2890  2           }

2892  2           previous_state = *state_ptr;
2893  2           *state_ptr = *next_state_ptr;

2895  2           switch (*state_ptr)
2896  3           {
2897  3           case INSTATE_SEARCH_PREFIX0:
2898  3               if (debugmode)
2899  4               {
2900  4                 rbe_log_stats(0, "AUXPROC: (
                                      %c) INSTATE_SEARCH_PREFIX0\n", c);
2901  3               }
```



```
2903  3               if (c == REMFD_MAGIC_PREFIX[0])
2904  4               {
2905  4                   *next_state_ptr = INSTATE_SEARCH_PREFIXN;
2906  4                   n = 1;
2907  4                   *parsePos = 1;
2908  3               }
2909  3               break;

2911  3           case INSTATE_SEARCH_PREFIXN:
2912  3               if (debugmode)
2913  4               {
2914  4                 rbe_log_stats(0, "AUXPROC: (
                                        %c) INSTATE_SEARCH_PREFIXN\n", c);
2915  3               }

2917  3               if (c != REMFD_MAGIC_PREFIX[n])
2918  4               {
2919  4                   *next_state_ptr = INSTATE_SEARCH_PREFIX0;
2920  3               }
2921  3               else
2922  4               {
2923  4                 n++;
2924  4                 (*parsePos)++;
2925  4                 if (n == REMFD_MAGIC_LENGTH)
2926  5                 {
2927  5                   *next_state_ptr = INSTATE_GATHER_COOKIE;
2928  5                   n = 0;
2929  5                   *parsePos = 0;
2930  4                 }
2931  3               }
2932  3               break;

2934  3           case INSTATE_GATHER_COOKIE:
2935  3               if (debugmode)
2936  4               {
2937  4                 rbe_log_stats(0, "AUXPROC: (
                                        %c) INSTATE_GATHER_COOKIE\n", c);
2938  3               }

2940  3               if (c == REMFD_MAGIC_SUFFIX[0])
2941  4               {
2942  4                   /*
2943  4                    * Got the cookie
2944  4                    */

2946  4                   cookiebuf[n] = '\0';
2947  4                   *next_state_ptr = INSTATE_SEARCH_SUFFIXN;
2948  4                   n = 1;
2949  4                   *parsePos = 1;
2950  3               }
2951  3               else if (strchr(REMFD_COOKIE_CHARS, c) == NULL)
2952  4               {
2953  4                   /*
2954  4                    * We found a valid prefix (else we would
2955  4                    * not be here) but the cookie contains an
2956  4                    * illegal character.  Should not happen;
2957  4                    * there has been some protocol failure.
2958  4                    */

2960  4                   *exitp = SPEXIT_REMOTE_STDERR_FAIL;
2961  4                   ret_status = TRUE;
2962  4                   protfailed = 1;
2963  3               }
2964  3               else if (n == REMFD_MAX_COOKIE_LEN)
2965  4               {
```

```
2966  4                 /*
2967  4                  * Should have seen the SUFFIX[0] by now.
2968  4                  */

2970  4                 *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
2971  4                 ret_status = TRUE;
2972  4                 protfailed = 1;
2973  3             }
2974  3             else
2975  4             {
2976  4                 /*
2977  4                  * another cookie character.
2978  4                  */

2980  4                 cookiebuf[n] = c;
2981  4                 n++;
2982  4                 (*parsePos)++;
2983  3             }
2984  3             break;

2986  3         case INSTATE_SEARCH_SUFFIXN:
2987  3             if (debugmode)
2988  4             {
2989  4               rbe_log_stats(0, "AUXPROC: (
                                    %c) INSTATE_SEARCH_SUFFIXN\n", c);
2990  3             }

2992  3             if (c != REMFD_MAGIC_SUFFIX[n])
2993  4             {
2994  4                 *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
2995  4                 ret_status = TRUE;
2996  4                 protfailed = 1;
2997  3             }
2998  3             else
2999  4             {
3000  4               n++;
3001  4               (*parsePos)++;
3002  4               if (n == REMFD_MAGIC_LENGTH)
3003  5               {
3004  5                 *next_state_ptr = INSTATE_NEWLINE;
3005  4               }
3006  3             }
3007  3             break;

3009  3         case INSTATE_GATHER_STATUS:
3010  3             if (debugmode)
3011  4             {
3012  4                rbe_log_stats(0, "AUXPROC: (
                                    %c) INSTATE_GATHER_STATUS\n", c);
3013  3             }

3015  3             if (isdigit(c))
3016  4             {
3017  4                 char xx[2];

3019  4                 xx[0] = c;
3020  4                 xx[1] = '\0';

3022  4                 temp_exit_status *= 10;
3023  4                 temp_exit_status += atoi(xx);
3024  4                 n++;
3025  4                 (*parsePos)++;
3026  4                 *skipping_leading_whitespace = FALSE;

3028  3             }
3029  3             else if (c == '\n')
```



```
3030  4             {
3031  4                 *exitp = temp_exit_status;
3032  4                 ret_status = TRUE;
3033  4                 done = 1;
3034  3             }
3035  3             else if (c == ' ' &&
3036  3                      ((n == 0) || *skipping_leading_whitespace ))
3037  4             {
3038  4               *skipping_leading_whitespace = TRUE;

3040  4                 /*
3041  4                  * no-op, skip leading whitespace
3042  4                  */
3043  3             }
3044  3             else
3045  4             {
3046  4                 *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
3047  4                 ret_status = TRUE;
3048  4                 protfailed = 1;
3049  3             }
3050  3             break;

3052  3         case INSTATE_COPY_TO_STDOUT:

3054  3             /*
3055  3              * Eventually, the protocol should include
3056  3              * an explicit length for this state, so
3057  3              * we can do large read/writes and so we will not
3058  3              * be vulnerable to confusion based on gunk being
3059  3              * copied to stdout.
3060  3              *
3061  3              * For now, just shove the characters at stdout
3062  3              * and stop as soon as we fix PREFIX[0]
3063  3              */

3065  3             if (c != REMFD_MAGIC_PREFIX[0])
3066  4             {
3067  4                 /*
3068  4                  * build a buffer to write to the restore log
3069  4                  */
3070  4                 if (('\n' == c) || ((*msgPos) >= MSGBUFFLEN))
3071  5                 {
3072  5                   msgLogBuff[*msgPos] = '\0';
3073  5                   (void) rbe_log_stats(0,
3074  5                                        "%s: %s",
3075  5                                        remhostname,
3076  5                                        msgLogBuff);
3077  5                   if (write_prog)
3078  6                   {
3079  6                     WriteStringMsg(prog_fd,
3080  6                                    EDMREPROGMSG_RESTORE_RCMD_UNKNOWN,

3081  6                                    0,
3082  6                                    msgLogBuff);
3083  5                   }

3085  5                   memset(msgLogBuff, 0, MSGBUFFLEN + 1);
3086  5                   *msgPos = 0;
3087  4                 }
3088  4                 else
3089  5                 {

3091  5                    msgLogBuff[*msgPos] = c;
3092  5                    (*msgPos)++;
3093  4                 }
```

```
3095  3                 }
3096  3                 else
3097  4                 {
3098  4                     n = 1;
3099  4                     *parsePos = 1;
3100  4                     *next_state_ptr = INSTATE_SEARCH_PREFIXN;
3101  3                 }
3102  3                 break;

3104  3             case INSTATE_NEWLINE:
3105  3                 if (c == '\n')
3106  4                 {
3107  5                     if (previous_state == INSTATE_SEARCH_SUFFIXN) {
3108  5                         *next_state_ptr = decodecookie(cookiebuf);

3110  5                         memset(cookiebuf, 0, REMFD_MAX_COOKIE_LEN+1);

3112  5                         n = 0;
3113  5                         *parsePos = 0;
3114  4                     }
3115  4                     else
3116  5                     {
3117  5                         *next_state_ptr = INSTATE_SEARCH_PREFIX0;
3118  4                     }
3119  3                 }
3120  3                 break;
3121  2             }   /* end of switch */
3122  1         }   /* end of while loop */

3124  1         return ret_status;

3126      }         /* end of parse_remote_stderr_info2() */
```



```
3131      /*
3132       * ForwardXcpiogenProgress()
3133       *  REP: Restore Engine Process, AP: auxproc, XC: xcpiogen.
3134       *  This routine read progress messages from XC and forwards
3135       *  then to REP. This routine should not block on a read().
3136       *
3137       * Args:
3138       *  int xcpiogen_prog_fd -- the progres channel from XC to AP.
3139       *  int restore_engine_prog_fd -- the progress channel from AP to REP.
3140       *
3141       * Returns:
3142       *  int -- the number of forwarded messages. -1 for an error.
3143       *
3144       * Side Effects:
3145       *  Read and writes file descpitors.
3146       */
3147      static int
3148      ForwardXcpiogenProgress(int xcpiogen_prog_fd,
3149                              int restore_engine_prog_fd,
3150                              boolean_ty *zero_byte_read)
3151  1   {
3152  1     int bytes_skipped = 0;
3153  1     int read_ret, write_ret;
3154  1     int fd_test;
3155  1     int msg_count = 0;
3156  1     prog_chan_msg xcpiogen_msg;
3157  1     int debug_prog = 0;
3158  1     int write_prog = 1;

3160  1     *zero_byte_read = FALSE;

3162  1     memset(&xcpiogen_msg, 0, sizeof(prog_chan_msg));

3164  1     while(1 == (fd_test = fd_avail_test1(xcpiogen_prog_fd)))
3165  2     {
3166  2       read_ret =  ReadMsg(
                          xcpiogen_prog_fd, &xcpiogen_msg, &bytes_skipped);

3168  2       if(0 != bytes_skipped)
3169  3       {
3170  3         (void) rbe_log_stats(0,
3171  3                              "The progress channel from xcpiogen had %d
                                                              extranous bytes.");
3172  2       }
3173  2       if(0 == read_ret)
3174  3       {
3175  3         *zero_byte_read = TRUE;
3176  3         break;
3177  2       }
3178  2       else if(-1 == read_ret)
3179  3       {
3180  3         break;
3181  2       }
3182  2       if(debug_prog)
3183  3       {
3184  3         (void) rbe_log_stats(0,
3185  3                              "%s: %s",
3186  3                              "debug progress",
3187  3                              DEBUG_PROG_MESSAGE_PEAK(&xcpiogen_msg));
3188  2       }
3189  2       if(write_prog)
3190  3       {
```

```
3191  3         write_ret =  WriteMsg(restore_engine_prog_fd, &xcpiogen_msg);
3192  2       }
3193  2       msg_count++;
3194  2       free(xcpiogen_msg.pcm_body);

3196  2       memset(&xcpiogen_msg, 0, sizeof(prog_chan_msg));
3197  1     }
3198  1     if(-1 == fd_test)
3199  2     {
3200  2       (void) rbe_log_stats(RBRECOVER_MKERR(errno),
3201  2                            "Encountered error testing xcpiogen file
                                                                descriptor.");

3202  2       return -1;
3203  1     }
3204  1     return msg_count;
3205      }
```



```
3207    /*
3208     * static int DemuxAuxChildren()
3209     *
3210     * This function handles IPC communications between the following
3211     * processes:  REP: Restore Engine Process, AP: auxproc, XC: xcpiogen,

3212     * and RC: remote command. The RC sends error and warning messages
3213     * tp AP. When the RP is finished, RC's exit status is sent last.
3214     * The remote exit status indicates that the restore is finished.
3215     * RP error and warning messages are logged and forward on as
3216     * progress messages to RP. At the same time XC will send progress
3217     * informaton to AP, and AP will forward them to REP. XC does its
3218     * own logging.
3219     *
3220     * Args:
3221     * int progress_fd -- (In) this file descriptor is from AP to REP.
3222     * int remote_fd --  (In) this file descriptor is from RC to AP.
3223     * char *remote_progname -- (In) this the RP executable name.
3224     * int xcpiogen_fd -- (In) this file descriptor is from XC to AP.
3225     * int xcpiogen_pid -- (In) the pid for XC.
3226     * int *remote_exit -- (Out) the remote exit interperated.
3227     *
3228     * REP: Restore Engine Process, AP: auxproc, XC: xcpiogen,
                                                          RC: remote command.
3229     *
3230     * Returns: ??
3231     *
3232     */

3234    static int
3235    DemuxAuxChildren(int progress_fd,
3236                     int remote_fd,
3237                     char *remote_progname,
3238                     int xcpiogen_fd,
3239                     int xcpiogen_pid,
3240                     int *remote_exit)
3241  1 {

3243  1         int remote_exitinfo;
3244  1         int forward_ret;
3245  1         boolean_ty remote_exitted = FALSE;
3246  1         struct pollfd monitor_fds[2];
3247  1         int poll_ret;
3248  1         boolean_ty remote_first_call = TRUE;
3249  1         int logging_channel = 0;
3250  1         int start_ec = attn_ec; /* attn_ec global */
3251  1         boolean_ty xcpiogen_zero_byte_reads;
3252  1         int number_fds;
3253  1         /* The below  args are needed by and maintained by
3254  1          * parse_remote_stderr_info2(). No need to initialize
3255  1          * or test them, they simply maintain state information
3256  1          * for multiple invocations of parse_remote_stderr_info2().
3257  1          */

3259  1         enum input_states remote_state_ptr;
3260  1         enum input_states remote_next_state_ptr;
3261  1         boolean_ty skip_whitespace;
3262  1         int parsePos;
3263  1         int MsgPos;

3265  1         number_fds = 2;
3266  1         monitor_fds[0].fd = remote_fd;
3267  1         monitor_fds[0].events = POLLIN;
3268  1         monitor_fds[0].revents = 0;
```

```
3270 1          monitor_fds[1].fd = xcpiogen_fd;
3271 1          monitor_fds[1].events = POLLIN;
3272 1          monitor_fds[1].revents = 0;

3274 1          while((FALSE == remote_exitted) &&
3275 1                ((0 == ( poll_ret = CDL_poll_read(
                                    monitor_fds, number_fds, 5000))) ||
3276 1                (poll_ret > 0) ||
3277 1                 ((EINTR == errno) && (-1 == poll_ret) && (
                                              start_ec != attn_ec))))
3278 2          {
3279 2            if (0 == poll_ret)
3280 3            {
3281 3              /* timed out */
3282 3              /* check for the attention signal and other periodic
                                                                 checks. */
3283 3              if (start_ec != attn_ec)
3284 4              {
3285 4                /* Timeout on read, AND we were canceled (SIGUSR1) --
3286 4                   If this is an SSL socket read timeout, the client may
3287 4                   have gone away without us knowing it,
                                              so break out of the loop
3288 4                   without the remote exit status.
                                              Do this for all remote
3289 4                    remote process connections,
                                          since the xcpiogen to recxcpio
3290 4                    connection might be SSL,
                                          but not the remote_fd. In this case
3291 4                    the remote process will not know that xcpiogen is
                                                                    gone.
3292 4                 */

3294 4                 *remote_exit = SPEXIT_REMOTE_NO_STATUS;
                                          /* leaving w/o status */

3296 4                 rbe_user_error (0,
3297 4                                 "No remote exit status received in 5
                                              seconds -- stopped "
3298 4                                 "waiting for aborted restore's remote
                                              process to finish");
3299 4                  break;

3301 3              }
3302 2            }
3303 2            else if (-1 == poll_ret)
3304 3            {  /* eintr */
3305 3              /* log this here */
3306 3              /*rbe_log_stats(errno, "auxproc -- ");*/
3307 2            }
3308 2            else
3309 3            {
3310 3              if((POLLRDNORM & monitor_fds[1].revents) ||
3311 3                 (POLLRDBAND & monitor_fds[1].revents) ||
3312 3                 (POLLIN & monitor_fds[1].revents) ||
3313 3                 (POLLHUP & monitor_fds[1].revents))
3314 4              {
3315 4                /* Xcpiogen process is expected to send progress
                                                              formatted
3316 4                 * messages,
                            including warnings and errors along with restore
3317 4                 * progress over its stderr channel.
                                          We will check xcpiogen
3318 4                 * prior to the remote channel.
                                       This presumes that the remote
3319 4                 * channel determines whether we are done with the
                                                              restore.
```



```
3320 4                  */
3321 4                 monitor_fds[1].revents = 0;

3323 4                 xcpiogen_zero_byte_reads = FALSE;

3325 4                 forward_ret = ForwardXcpiogenProgress(
                                                        xcpiogen_prog_fd,
3326 4                                                   progress_fd,
3327 4
                                                &xcpiogen_zero_byte_reads);
3328 4                 if(TRUE == xcpiogen_zero_byte_reads)
3329 5                 {
3330 5                   /* Lets no longer wait on this fd,
3331 5                    * if we get a zero byte read. The
3332 5                    * second fd is for xcpiogen.
3333 5                    */
3334 5                   number_fds = 1;
3335 4                 }
3336 3               }
3337 3               if((POLLRDNORM & monitor_fds[0].revents) ||
3338 3                  (POLLRDBAND & monitor_fds[0].revents) ||
3339 3                  (POLLIN & monitor_fds[0].revents) ||
3340 3                  (POLLHUP & monitor_fds[0].revents))
3341 4               {
3342 4                 /* POLLRDBAND Why do we need these conditions. */

3344 4                 /* Remote process is expected to send information back
                                                              on stderr
3345 4                  * stream.  Read that information, parse it,
                                                        and if available
3346 4                  * send remote exit status back to parent.
                                                        If the exit status
3347 4                  * is read this loop is terminated.
3348 4                  */
3349 4                 monitor_fds[0].revents = 0;

3351 4                 remote_exitted = parse_remote_stderr_info2(progress_fd,
3352 4                                                            remote_fd,
3353 4
                                                        &remote_exitinfo,
3354 4
                                                          remote_progname,
3355 4
                                                        remote_first_call,
3356 4
                                                        &remote_state_ptr,
3357 4
                                                    &remote_next_state_ptr,
3358 4
                                                         &skip_whitespace,
3359 4                                                         &parsePos,
3360 4                                                         &MsgPos);
3361 4                 remote_first_call = FALSE;

3362 3               }
3363 2             }
3364 1           }

3366 1           /* Should we test xcpiogen for a last time before we return!
                                                                        */

3368 1           if(1 == fd_avail_test(xcpiogen_prog_fd))
3369 2           {
3370 2             xcpiogen_zero_byte_reads = FALSE;

3372 2             forward_ret = ForwardXcpiogenProgress(xcpiogen_prog_fd,
```

```
3373  2                                                progress_fd,
3374  2
                                                     &xcpiogen_zero_byte_reads);
3375  1          }

3377  1          if (-1 == poll_ret)
3378  2          {  /* Not eintr */
3379  2            rbe_log_stats(RBRECOVER_MKERR(errno),
3380  2                          "Auxproc failed to poll Children pids!");
3381  2            return -1;
3382  1          }
3383  1          if(TRUE == remote_exitted)
3384  2          {
3385  2            *remote_exit = remote_exitinfo;
3386  2            return 0;
3387  1          }

3389    } /* DemuxAuxChildren() */
```

```
  1  /*
  2   *                  RSLauxsupp.c
  3   *
  4   * Copyright (C) 1998 by EMC Corp, Inc.  All rights reserved.
  5   *
  6   *
  7   * Table of Contents:
  8   *--------------------
  9  ** int looprw(int fd, char *buf, int nbytes, int (*func)());
 10  **
 11   */


 14  #define  _POSIX_SOURCE 1
 15  #include <eb/eb_port.h>



 19  #include <netdb.h>
 20  #include <pwd.h>
 21  #include <ctype.h>
 22  #include <sys/wait.h>
 23  #include <sys/types.h>
 24  #include <util/esl_select.h>
 25  #include <eerrno/esl_strerror.h>
 26  #include <util/esl_limit.h>
 27  #include <stdlib.h>


 30  #include <ebconfig/rbconfig.h>
 31  #include <epffdb/ffdb.h>
 32  #include <epffdb/rbfinclient.h>
 33  #include <ebutil/ebsock_if.h>
 34  #include <eb/rb_log.h>

 36  #include <RSLinterns.h>
 37  #include <RSLspexits.h>
 38  #include <RSLremfd.h>
 39  #include "RSLauxSupp.h"


 42  static char *oops_msg = NULL;
 43  static char oops_string[1024];


 46  /*
 47   * read() or write() a file descriptor, looping as
 48   * necessary until all bytes are obtained.  Useful for reading
 49   * from pipes or sockets, or anything else that is permitted to
 50   * return fewer bytes than you ask for.
 51   */

 53  int
 54  looprw(int fd,
 55         char *buf,
 56         int nbytes,
 57         int (* func)(int fd, char *buf, int nbytes))
 58 1{
 59 1    int todo;

 61 1    if (nbytes <= 0)            /* (in)sanity */
 62 2    {
 63 2        return nbytes;
 64 1    }

 66 1    /*
```



```
 67 1     * Each time try to ''func'' the entire amount remaining.
 68 1     * Exit the loop successfully when we've done it all.
 69 1     * If ''func'' returns an error, propogate the error.
 70 1     * If ''func'' returns EOF, return the amount done
 71 1     * prior to the EOF.
 72 1     */

 74 1    todo = nbytes;
 75 1    for (;;)
 76 2    {
 77 2        int r;

 79 2        r = (* func)(fd, buf, todo);
 80 2        if (r == todo)                  /* all done */
 81 3        {
 82 3            return nbytes;
 83 2        }

 85 2        else if (r == 0)                /* EOF */
 86 3        {
 87 3            return nbytes - todo;       /* amt done before EOF */
 88 2        }

 90 2        else if (r == -1)               /* error */
 91 3        {
 92 3            return -1;
 93 2        }

 95 2        buf += r;
 96 2        todo -= r;
 97 1    }

 99 1    /*NOTREACHED*/
100 }       /* end of looprw() */
```

```
104  /*
105   * Use these functions with looprw to build
106   * a loop read (or loop write) that ignores EINTR.
107   */
108
109  int
110  read_no_eintr(int fd,
111                char *buf,
112                int nbytes)
113 1{
114 1      int r;
115
116 1      do
117 2      {
118 2          errno = 0;
119 2          r = read(fd, buf, (uint_t)nbytes);
120 1      } while (r == -1 && errno == EINTR);
121
122 1      return r;
123  }        /* end of read_no_eintr() */
```



```
125  int
126  write_no_eintr(int fd,
127                 char *buf,
128                 int nbytes)
129 1{
130 1      int r;
131
132 1      do
133 2      {
134 2          errno = 0;
135 2          r = write(fd, buf, (uint_t)nbytes);
136 1      } while (r == -1 && errno == EINTR);
137
138 1      return r;
139  }        /* end of write_no_eintr() */
```

```
141  /*
142   * Read bytes per protocol.  Filter out zero-length reads.
143   * Die if we don't all the bytes we are supposed to get.
144   */

146  void
147  pread_or_die(int fd,
148                char *buf,
149                int nbytes,
150                void (* diefunc)(int))
151 1 {
152 1      if (nbytes > 0)
153 2      {
154 2           if (looprw(fd, buf, nbytes, read_no_eintr) != nbytes)
155 3           {
156 3                sprintf(
                          oops_string, "Unable to read %d bytes from fd %d%s",
157 3                        nbytes, fd, (0 == errno) ? ", end of file" : "");
158 3                oops_msg = oops_string;
159 3                (* diefunc)(33);
160 2           }
161 1      }
162  }         /* end of pread_or_die() */
```



```
165  /*
166   * Read bytes per protocol.  Filter out zero-length reads.
167   * Die if we don't all the bytes we are supposed to get.
168   */

170  int
171  pread_or_warn(int fd,
172                char *buf,
173                int nbytes,
174                int (* comm_warning_func)(int))
175 1 {
176 1      if (nbytes > 0)
177 2      {
178 2           if (looprw(fd, buf, nbytes, read_no_eintr) != nbytes)
179 3           {
180 3                sprintf(
                          oops_string, "Unable to read %d bytes from fd %d%s",
181 3                        nbytes, fd, (0 == errno) ? ", end of file" : "");
182 3                oops_msg = oops_string;
183 3                (* comm_warning_func)(33);
184 3                return -1;
185 2           }
186 2           return nbytes;
187 1      }
188  }         /* end of pread_or_warn() */
```

```
194   /*
195    * same, but write
196    */
198   void
199   pwrite_or_die(int fd,
200                 char *buf,
201                 int nbytes,
202                 void (* diefunc)(int))
203 1 {
204 1     if (nbytes > 0)
205 2     {
206 2          if (looprw(fd, buf, nbytes, write_no_eintr) != nbytes)
207 3          {
208 3              sprintf(oops_string, "Unable to write %d bytes to fd %d",
209 3                      nbytes, fd);
210 3              oops_msg = oops_string;
211 3              (* diefunc)(34);
212 2          }
213 1     }
214   }         /* end of pwrite_or_die() */
```



```
217   /*
218    * same, but write
219    */
221   int
222   pwrite_or_warn(int fd,
223                  char *buf,
224                  int nbytes,
225                  int (* comm_warning_func)(int))
226 1 {
227 1     if (nbytes > 0)
228 2     {
229 2          if (looprw(fd, buf, nbytes, write_no_eintr) != nbytes)
230 3          {
231 3              sprintf(oops_string, "Unable to write %d bytes to fd %d",
232 3                      nbytes, fd);
233 3              oops_msg = oops_string;
234 3              (* comm_warning_func)(34);
235 3              return -1;
236 2          }
237 2          return nbytes;
238 1     }
239   }         /* end of pwrite_or_warn() */
```

```
241     /*
242      * Send a command packet down an auxproc file descriptor.
243      * Returns number of bytes sent or -1 for unrecoverable error.
244      */
246     int
247     auxcmdpacket(int fd,
248                  char cmd,
249                  int datalen,
250                  char *data)
251   1 {
252   1     int retStatus1 = 0;
253   1     int retStatus2 = 0;
254   1     int retStatus3 = 0;
255   1     retStatus1 = pwrite_or_warn(fd, &cmd, 1, auxproc_comm_warning);
256   1     retStatus2 = pwrite_or_warn(fd, (char *)&datalen, sizeof datalen,
257   1                                 auxproc_comm_warning);
258   1     retStatus3 = pwrite_or_warn(
                                fd, data, datalen, auxproc_comm_warning);
259   1     if ((-1 ==  retStatus1) || (-1 ==  retStatus2) || (
                                                  -1 ==  retStatus3) )
260   2     {
261   2       return -1;
262   1     }
263   1     return (retStatus1 + retStatus2 + retStatus3);
265     }         /* end of auxcmdpacket() */
```



```
267     /*
268      * Common portion of auxresults/auxresults_intr
269      */
271     int
272     auxres2(int     fd,
273             char    cmd,
274             int     fixedbuflen,
275             char  **fixedbufp)
276   1 {
277   1     int datalen;
278   1     int retStatus1 = 0;
279   1     int retStatus2 = 0;
281   1     retStatus1 = pread_or_warn(fd, (char *)&datalen, sizeof datalen,
282   1                                auxproc_comm_warning);
284   1     if (datalen > fixedbuflen)
285   2     {
286   2         if ((*fixedbufp = (char *) malloc(datalen)) == NULL)
287   3         {
288   3             rbe_log_stats(0, "Could not allocate memory in auxres2");
289   3             return (-1);
290   2         }
291   1     }
293   1     retStatus2 = pread_or_warn(fd, *fixedbufp, datalen,
294   1                                auxproc_comm_warning);
296   1     if ((-1 ==  retStatus1) || (-1 ==  retStatus2))
297   2     {
298   2       return -1;
299   1     }
301   1     return datalen;
302     }         /* end of auxres2() */
```

```
305  /*
306   * Read a reply packet beginning with 'cmd';
307   * return results via the buffer passed in fixedbufp,
308   * if it is large enough, else allocate a new buffer.
309   *
310   * Returns length of returned results.  Fails with -1;
311   */

313  int
314  auxresults(int fd,
315             char cmd,
316             int fixedbuflen,
317             char **fixedbufp)
318 1 {
319 1      char c;
320 1      int retStatus1 = 0;
321 1      int retStatus2 = 0;

323 1      retStatus1 = pread_or_warn(fd, &c, 1, auxproc_comm_warning);

325 1      if (c != cmd)
326 2      {
327 2           sprintf(
                          oops_string, "auxresults: expected command '%c',
                                                  received command '%c'",
328 2                  cmd, c);
329 2           oops_msg = oops_string;
330 2           errno = 0;
331 2           auxproc_comm_warning(65);

333 2           /* flush rest of message, if any */
334 2           retStatus2 = auxres2(fd, cmd, fixedbuflen, fixedbufp);

336 2           return -1;              /* report failure */
337 1      }
338 1      if (-1 ==  retStatus1)
339 2      {
340 2        return -1;
341 1      }

343 1      retStatus2 = auxres2(fd, cmd, fixedbuflen, fixedbufp);
344 1      return retStatus2;
345  }         /* end of auxresults() */
```



```
350  /* This should not remain an exit, or should it */
351  /* fatal_auxproc should be; indicate unrecoverable
352  auxproc communication error */

354  /* Remove use of stderr */
355  /* Pass oops string as input.*/

357  int
358  auxproc_comm_warning(int tracer_nbr)
359 1 {
360 1      char *errno_msg = NULL;

362 1      if (0 != errno)
363 2      {
364 2           errno_msg = esl_strerror(errno);
365 1      }
366 1      rbe_log_stats(0,

368 1                    "%s: UNRECOVERABLE auxproc communication, (
                                                  tracer #%d), %s, %s (%d)\n",
369 1                    progname, tracer_nbr,
370 1                    (NULL != oops_msg)  ? oops_msg  : "",
371 1                    (NULL != errno_msg) ? errno_msg : "", errno);
372 1      return(tracer_nbr);
373  }         /* end of auxproc_comm_warning() */
```

```
375  /*
376   * Wait, interruptibly,
                          for at least one byte to become available on fd.
377   *
378   * Returns 1 if at least one byte is available.
379   * Returns 0 if no bytes are available.
380   * Returns -1 for any type of failure, including wait interruption.
381   * Sets errno appropriate when -1 is returned.
382   */

384  int
385  fd_avail_test(int fd)
386 1 {
387 1      esl_fdset_ty rdbits;
388 1      struct esl_timeval timeout = { 0 , 0 };
389 1      int ret_status;

391 1      E_FD_ZERO(&rdbits);
392 1      E_FD_SET(fd, &rdbits);

394 1      /*
395 1       * Don't need to examine rdbits after select, since only one
396 1       * fd is in the set -- therefore return value can be computed
397 1       * directly from select return value.
398 1       */

400 1      ret_status = esl_select(
                           E_FD_SETSIZE, &rdbits, NULL, NULL, &timeout);

402 1      return ret_status;

404  }         /* end of fd_avail_test() */
```



```
409  /*
410   * ChildDone()
411   *
412   * Description:
413   *  Check for the child being done. If the EINTR error
414   *  is encountered for the waitpid then it will be
415   *  retry.
416   *
417   *  Remember that the child will become a defunct process
418   *  if its parent process does not wait on it.
419   *
420   * Errno:
421   * NOT ECHILD pid not running or not in group. This is handled
422   * by reset errno to 0 and returning 5.
423   *
424   * EINVAL Invalid argument.
425   * ERANGE The return of the child from waitpid was not expacted.
426   *
427   * Parameters:
428   *  (I) child_pid -- child pid to check.
429   *  (O) child_result -- childs exit status or signal.
430   *                      (determined by return value)
431   *
432   * Returns: int
433   *  -1 internal error or system error. errno is set.
434   *  0  child still running.
435   *  1  child exited. child_result is exit code.
436   *  2  child signalled (no core), child_result is signal.
437   *  3  child signalled core file generated,
                                  child_result is signal. *NOT Supported *
438   *  4  child stopped. child_result is signal.
439   *  5  child does not exist or is not a child of the calling process.
440   *
441   * Side effects:
442   *  errno can be reset waitpid OR ChildDone.
443   */
444  /* HACK */

446  int ChildDone(int child_pid, int *child_result)
447 1 {
448 1   int child_exit_val = 0;
449 1   int wait_status = 0;

451 1   if (NULL == child_result)
452 2   {
453 2     errno = EINVAL;
454 2     rbe_log_stats(RBRECOVER_MKERR(errno),
455 2                   "Can't waitpid for child,
                                        ChildDone:invalid parameters.\n");

457 2     return -1;

459 1   }

461 1   *child_result = 0;

463 1 #if 0
464 1       while((0 < (wait_status = waitpid(
                                  child_pid, &child_exit_val, WNOHANG))) &&
465 1             eintr_retries > 0)
466 2       {
467 2          if (EINTR == errno)
```

```
468 3           {
469 3             eintr_retries--;
470 2           }
471 2           else if(ECHILD == errno)
472 3           {
473 3             *child_result = 0;
474 3             errno = 0;
475 3             return 5;
476 2           }
477 2           else
478 3           {
479 3             /*
480 3              * Only retry on EINTR
481 3              */
482 3             break;
483 2           }
484 1         }
485 1 #endif

487 1       /*
488 1        * Do a waitpid NOHANG.
                          Retry EINTR and return internal for other errors.
489 1        * Set ECHILD to return 5.
                                  This indicates the child process is no
490 1        * longer around.
491 1        */

493 1       do
494 2       {
495 2          wait_status = waitpid(child_pid, &child_exit_val, WNOHANG);

497 1       } while((-1 == wait_status) && (EINTR == errno));


500 1       switch(wait_status)
501 2       {
502 2         case(-1): /* error encountered */

504 2           if(ECHILD == errno)
505 3           {
506 3             /* Lets handle the no child case as not an internal error.
                                                                        */
507 3             *child_result = 0;
508 3             errno = 0;
509 3             return 5;
510 2           }
511 2           rbe_log_stats(RBRECOVER_MKERR(errno),
512 2                         "Can't waitpid for child, error %s == %d\n",
513 2                          strerror(errno), errno);

515 2           return -1;

517 2         case(0): /* child still running */

519 2           return 0;

521 2         default: /* child is NOT running */

523 2           if (WIFEXITED(child_exit_val))
524 3           {
525 3             *child_result = WEXITSTATUS(child_exit_val);
526 3             return 1;
527 2           }
528 2           else if (WIFSIGNALED(child_exit_val))
529 3           {
530 3             *child_result = WTERMSIG(child_exit_val);
```



```
531 3 #if 0
532 3               /* WCOREDUMP is not _POSIX_C_SOURCE */
533 3               if(WCOREDUMP(child_exit_val))
534 3                 return 3;
535 3               else
536 3                 return 2;
537 3 #else
538 3               return 2;
539 3 #endif
540 2             }
541 2             else if (WIFSTOPPED(child_exit_val))
542 3             {
543 3               *child_result = WSTOPSIG(child_exit_val);
544 3               return 4;
545 2             }
546 2             else
547 3             {
548 3               errno = ERANGE;
549 3               rbe_log_stats(RBRECOVER_MKERR(errno),
550 3                             "Can't determine waitpid status of the
                                                                child process.\n");
551 3               return -1;
552 2             }
553 1         }
554   } /* End ChildDone() */
```